Subject: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by serue on Fri, 08 Jun 2007 15:54:37 GMT
View Forum Message <> Reply to Message

As I mentioned earlier, I don't know what sort of approach we want
to take to guide checkpoint and restart. I.e. do we want it to be
a mostly userspace-orchestrated affair, or entirely done in the
kernel using the freezer or some other mechanism in response to a
single syscall or containerfs file write?

If we wanted to do a lot of the work in userspace, here is a pair of
patches to read and restore signal information. It's entirely unsafe
wrt locking, bc i would assume that if we did in fact do c/r from
userspace, we would have some way of entirely pulling the task off
the runqueue while doing our thing...

Anyway, this is purely to start discussion.

thanks,
-serge

>From 30bbe322942e5ed86bad63861dad80595cd04063 Mon Sep 17 00:00:00 2001
From: Serge E. Hallyn <serue@us.ibm.com>
Date: Mon, 30 Apr 2007 16:22:44 -0400
Subject: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/

Define /proc/<pid>/sig/ directory containing files to report
on a process' signal info.

Files defined:
 action: list signal action
 altstack: print sigaltstack location and size
 blocked: print blocked signal mask
 pending: print pending signals and siginfo
 shared_pending: print shared pending signals and siginfo
 waiters: list tasks wait4()ing on task PID.

TESTING:
In one terminal run 'forker' as compiled from forker.c below. In
another terminal, run 'sh checkforker.sh' also included below
which will send more signals to the forker, then check it's pending,
blocked, and shared_pending sig/ files.

==========
forker.c
==========

void runchild(void)

```
{
 printf("child exiting\n");
 exit(0);
}

int main()
{
 sigset_t blockset;
 sigset_t prevset;
 int pid;

 if (sigemptyset(&blockset) == -1) {
  perror("sigemptyset");
  return -1;
 }

 if (sigaddset(&blockset, SIGUSR1) == -1) {
  perror("sigaddset");
  return -1;
 }

 if (sigaddset(&blockset, SIGUSR2) == -1) {
  perror("sigaddset");
  return -1;
 }

 if (sigaddset(&blockset, SIGCHLD) == -1) {
  perror("sigaddset");
  return -1;
 }

 if (sigaddset(&blockset, SIGSEGV) == -1) {
  perror("sigaddset");
  return -1;
 }

 if (sigaddset(&blockset, SIGPOLL) == -1) {
  perror("sigaddset");
  return -1;
 }

 if (sigprocmask(SIG_SETMASK, &blockset, &prevset) == -1) {
  perror("sigprocmask");
  return -1;
 }

 pid = fork();
 if (pid < 0)
```

```
  perror("fork");
 if (pid == 0)
  runchild();
 sleep(30);
}
```

```
checkforker.sh
```

```
p=`ps -ef | grep forker | head -1 | awk '{ print $2 }'`
kill -10 $p
kill -29 $p
kill -11 $p
echo pending
cat /proc/$p/sig/pending
echo blocked
cat /proc/$p/sig/blocked
echo shared pending
cat /proc/$p/sig/shared_pending
```

Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>

 Documentation/filesystems/proc.txt | 59 ++++++++++--
 fs/proc/base.c | 62 ++++++++++++
 include/linux/signal.h | 1 +
 kernel/signal.c | 202 ++
 4 files changed, 312 insertions(+), 12 deletions(-)

diff --git a/Documentation/filesystems/proc.txt b/Documentation/filesystems/proc.txt
index 8756a07..1d74e19 100644
--- a/Documentation/filesystems/proc.txt
+++ b/Documentation/filesystems/proc.txt
@@ -133,6 +133,7 @@ Table 1-1: Process specific entries in /proc
  maps Memory maps to executables and library files (2.4)
  mem Memory held by this process
  root Link to the root directory of this process
+ sig   Directory, which contains signal information
  stat Process status
  statm Process memory status information
  status Process status in human readable form
@@ -188,16 +189,50 @@ Table 1-2: Contents of the statm files (as of 2.6.8-rc3)
  dt   number of dirty pages   (always 0 on 2.6)
 ....

+Table 1-3: Contents of sig directory files
+..

+ File Content
+ action List sigaction as "signum flags sigmask handler"
+ altstack List location and size of sigaltstack
+ blocked For each signal, print '1' if blocked, '0' otherwise
+ pending For each signal, print '1' if pending for this thread,
+ '0' otherwise
+ Also print additional information for certain given
+ pending signals. See 'pending' entry below for details.
+ shared_pending For each signal, print '1' if pending for all threads,
+ '0' otherwise.
+ waiters For each task doing wait on task PID, list the flags and
+ the waiting process' pid.
+.....
+
+The /proc/PID/sig/pending file prints additional pending signal information
+depending on the signal. See Table 1-4 for details by signal number.
+
+Table 1-4: Details of /proc/PID/sig/pending info for pending signals
+.....
+ Signal Information printed
+ SIGKILL Pid and uid of killing process
+
+ SIGCHLD Pid, uid, status, utime, and stime of exited child process
+
+ SIGILL
+ SIGFPE
+ SIGSEGV
+ SIGBUS Address and (if applicable to arch) trap number
+
+ SIGPOLL Band and fd
+.....
+
1.2 Kernel data

Similar to the process entries, the kernel data files give information about
the running kernel. The files used to obtain this information are contained in
-/proc and are listed in Table 1-3. Not all of these will be present in your
+/proc and are listed in Table 1-5. Not all of these will be present in your
system. It depends on the kernel configuration and the loaded modules, which
files are there, and which are missing.

-Table 1-3: Kernel info in /proc
+Table 1-5: Kernel info in /proc
.....
File Content
apm Advanced power management info
@@ -476,7 +511,7 @@ subdirectories. These are named ide0, ide1 and so on. Each of

these
directories contains the files shown in table 1-4.

-Table 1-4: IDE controller info in /proc/ide/ide?
+Table 1-6: IDE controller info in /proc/ide/ide?
.....
File Content
channel IDE channel (0 or 1)
@@ -490,7 +525,7 @@ controllers directory. The files listed in table 1-5 are contained in these
directories.

-Table 1-5: IDE device information
+Table 1-7: IDE device information
.....
File Content
cache The cache
@@ -532,12 +567,12 @@ the drive parameters:
1.4 Networking info in /proc/net

-The subdirectory /proc/net follows the usual pattern. Table 1-6 shows the
+The subdirectory /proc/net follows the usual pattern. Table 1-8 shows the
additional values you get for IP version 6 if you configure the kernel to
-support this. Table 1-7 lists the files and their meaning.
+support this. Table 1-9 lists the files and their meaning.

-Table 1-6: IPv6 info in /proc/net
+Table 1-8: IPv6 info in /proc/net
.....
File Content
udp6 UDP sockets (IPv6)
@@ -552,7 +587,7 @@ Table 1-6: IPv6 info in /proc/net
.....

-Table 1-7: Network info in /proc/net
+Table 1-9: Network info in /proc/net
.....
File Content
arp Kernel ARP table
@@ -676,10 +711,10 @@ The directory /proc/parport contains information about the parallel
ports of
your system. It has one subdirectory for each port, named after the port
number (0,1,2,...).

-These directories contain the four files shown in Table 1-8.
+These directories contain the four files shown in Table 1-10.

-Table 1-8: Files in /proc/parport
+Table 1-10: Files in /proc/parport
 .....
  File   Content
  autoprobe Any IEEE-1284 device ID information that has been acquired.
@@ -697,10 +732,10 @@ Table 1-8: Files in /proc/parport

 Information about the available and actually used tty's can be found in the
 directory /proc/tty.You'll find entries for drivers and line disciplines in
-this directory, as shown in Table 1-9.
+this directory, as shown in Table 1-11.

-Table 1-9: Files in /proc/tty
+Table 1-11: Files in /proc/tty
 .....
  File   Content
  drivers   list of drivers and their usage
diff --git a/fs/proc/base.c b/fs/proc/base.c
index a5fa1fd..8e0dd7a 100644
--- a/fs/proc/base.c
+++ b/fs/proc/base.c
@@ -1681,6 +1681,66 @@ out_no_task:
  return ret;
 }

+/* /proc/<pid>/sig/ */
+static ssize_t proc_pid_sig_read(struct file * file, char __user * buf,
+   size_t count, loff_t *ppos)
+{
+ struct dentry *dentry = file->f_path.dentry;
+ struct inode * inode = dentry->d_inode;
+ char *p = NULL;
+ ssize_t length;
+ struct task_struct *task = get_proc_task(inode);
+
+ if (!task)
+ return -ESRCH;
+ length = task_read_procsig(task, (char*)dentry->d_name.name, &p);
+ put_task_struct(task);
+ if (length > 0)
+ length = simple_read_from_buffer(buf, count, ppos, p, length);
+ kfree(p);
+ return length;

```
+}
+
+static const struct file_operations proc_sig_attr_operations = {
+ .read  = proc_pid_sig_read,
+// .write  = proc_pid_sig_write,
+};
+
+
+static struct pid_entry sig_dir_stuff[] = {
+ REG("pending",           S_IRUGO|S_IWUGO, sig_attr),
+ REG("shared_pending",    S_IRUGO|S_IWUGO, sig_attr),
+ REG("blocked",           S_IRUGO|S_IWUGO, sig_attr),
+ REG("action",            S_IRUGO|S_IWUGO, sig_attr),
+ REG("waiters",           S_IRUGO|S_IWUGO, sig_attr),
+ REG("altstack",          S_IRUGO|S_IWUGO, sig_attr),
+};
+
+static int proc_sig_dir_readdir(struct file * filp,
+        void * dirent, filldir_t filldir)
+{
+ return proc_pident_readdir(filp,dirent, filldir,
+       sig_dir_stuff, ARRAY_SIZE(sig_dir_stuff));
+}
+
+static const struct file_operations proc_sig_dir_operations = {
+ .read  = generic_read_dir,
+ .readdir = proc_sig_dir_readdir,
+};
+
+static struct dentry *proc_sig_dir_lookup(struct inode *dir,
+    struct dentry *dentry, struct nameidata *nd)
+{
+ return proc_pident_lookup(dir, dentry,
+      sig_dir_stuff, ARRAY_SIZE(sig_dir_stuff));
+}
+
+static const struct inode_operations proc_sig_dir_inode_operations = {
+ .lookup  = proc_sig_dir_lookup,
+ .getattr = pid_getattr,
+ .setattr = proc_setattr,
+};
+
 #ifdef CONFIG_SECURITY
 static ssize_t proc_pid_attr_read(struct file * file, char __user * buf,
       size_t count, loff_t *ppos)
@@ -2006,6 +2066,7 @@ static const struct pid_entry tgid_base_stuff[] = {
 #ifdef CONFIG_TASK_IO_ACCOUNTING
  INF("io", S_IRUGO, pid_io_accounting),
```

```
 #endif
+ DIR("sig",        S_IRUGO|S_IXUGO, sig_dir),
 };

 static int proc_tgid_base_readdir(struct file * filp,
@@ -2286,6 +2347,7 @@ static const struct pid_entry tid_base_stuff[] = {
 #ifdef CONFIG_FAULT_INJECTION
  REG("make-it-fail", S_IRUGO|S_IWUSR, fault_inject),
 #endif
+ DIR("sig",        S_IRUGO|S_IXUGO, sig_dir),
 };

 static int proc_tid_base_readdir(struct file * filp,
diff --git a/include/linux/signal.h b/include/linux/signal.h
index 9a5eac5..6863543 100644
--- a/include/linux/signal.h
+++ b/include/linux/signal.h
@@ -243,6 +243,7 @@ struct pt_regs;
 extern int get_signal_to_deliver(siginfo_t *info, struct k_sigaction *return_ka, struct pt_regs *regs,
void *cookie);

 extern struct kmem_cache *sighand_cachep;
+extern int task_read_procsig(struct task_struct *p, char *name, char **value);

 /*
  * In POSIX a signal is sent either to a specific thread (Linux task)
diff --git a/kernel/signal.c b/kernel/signal.c
index 364fc95..a2a3ebe 100644
--- a/kernel/signal.c
+++ b/kernel/signal.c
@@ -2537,3 +2537,205 @@ void __init signals_init(void)
 {
  sigqueue_cachep = KMEM_CACHE(sigqueue, SLAB_PANIC);
 }
+
+/*
+ * print a sigset_t to a buffer.  Return # characters printed,
+ * not including the final ending '\0'.
+ */
+static int print_sigset(char *buf, sigset_t *sig)
+{
+ int i;
+
+ for (i=0; i<_NSIG; i++) {
+  if (sigismember(sig, i))
+   buf[i] = '1';
+  else
+   buf[i] = '0';
```

```
+ }
+ buf[_NSIG] = '\0';
+
+ return _NSIG;
+}
+
+static int print_sigset_alloc(char **bufp, sigset_t *sig)
+{
+ char *buf;
+
+ *bufp = buf = kmalloc(_NSIG+1, GFP_KERNEL);
+ if (!buf)
+  return -ENOMEM;
+
+ return print_sigset(buf, sig);
+}
+
+static int print_wait_info(char **bufp, struct signal_struct *signal)
+{
+ char *buf;
+ wait_queue_t *wait;
+
+ *bufp = buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
+ if (!buf)
+  return -ENOMEM;
+
+ spin_lock(&signal->wait_chldexit.lock);
+
+ list_for_each_entry(wait, &signal->wait_chldexit.task_list, task_list) {
+  struct task_struct *tsk = wait->private;
+
+  if (buf - *bufp +50 > PAGE_SIZE) {
+   spin_unlock(&signal->wait_chldexit.lock);
+   kfree(buf);
+   *bufp = NULL;
+   return -ENOMEM;
+  }
+  WARN_ON(wait->func != default_wake_function);
+  buf += sprintf(buf, "%u %d\n", wait->flags, tsk->pid);
+ }
+
+ spin_unlock(&signal->wait_chldexit.lock);
+
+ return buf-*bufp;
+}
+
+static int print_sigpending_alloc(char **bufp, struct sigpending *pending)
+{
```

```
+ int alloced=0;
+ char *buf, *p;
+ struct sigqueue *q;
+ struct siginfo *info;
+
+ alloced = PAGE_SIZE;
+ p = buf = kmalloc(alloced, GFP_KERNEL);
+ if (!buf)
+  return -ENOMEM;
+
+ p += print_sigset(buf, &pending->signal);
+ p += sprintf(p, "\n");
+
+ list_for_each_entry(q, &pending->list, list) {
+  info = &q->info;
+  if (p-buf+215 > alloced) {
+   int len=p-buf;
+   char *buf2;
+   alloced += PAGE_SIZE;
+   buf2 = kmalloc(alloced, GFP_KERNEL);
+   if (!buf2) {
+    kfree(buf);
+    return -ENOMEM;
+   }
+   memcpy(buf2, buf, alloced - PAGE_SIZE);
+   kfree(buf);
+   buf = buf2;
+   p = buf+len;
+  }
+
+  p += sprintf(p, "sig %d: user %d flags %d",
+   info->si_signo, (int)q->user->uid, q->flags);
+  p += sprintf(p, " errno %d code %d\n",
+   info->si_errno, info->si_code);
+
+  switch(info->si_signo) {
+  case SIGKILL:
+   p += sprintf(p, "  spid %d suid %d\n",
+    info->_sifields._kill._pid,
+    info->_sifields._kill._uid);
+   break;
+  /* XXX skipping posix1b timers and signals for now */
+  case SIGCHLD:
+   p += sprintf(p, "  pid %d uid %d status %d utime %lu stime %lu\n",
+    info->_sifields._sigchld._pid,
+    info->_sifields._sigchld._uid,
+    info->_sifields._sigchld._status,
+    info->_sifields._sigchld._utime,
```

```
+    info->_sifields._sigchld._stime);
+   break;
+  case SIGILL:
+  case SIGFPE:
+  case SIGSEGV:
+  case SIGBUS:
+#ifdef __ARCH_SI_TRAPNO
+   p += sprintf(p, "  addr %lu trapno %d\n",
+    (unsigned long)info->_sifields._sigfault._addr,
+    info->_sifields._sigfault._trapno);
+#else
+   p += sprintf(p, "  addr %lu\n",
+    (unsigned long)info->_sifields._sigfault._addr);
+#endif
+   break;
+  case SIGPOLL:
+   p += sprintf(p, "  band %ld fd %d\n",
+    (long)info->_sifields._sigpoll._band,
+    info->_sifields._sigpoll._fd);
+   break;
+  default:
+   p += sprintf(p, "  Unsupported siginfo for signal %d\n",
+    info->si_signo);
+   break;
+  }
+ }
+ *bufp = buf;
+ return p-buf;
+}
+
+static int print_sigaction_list(char **bufp, struct sighand_struct *sighand)
+{
+ struct k_sigaction *action;
+ int maxlen;
+ int i;
+ char *buf;
+
+ /* two unsigned longs (20 chars), one int (10 chars), a sigset_t, 3 spaces, plus a newline */
+ maxlen = 10 + 20*2 + _NSIG + 4;
+ /* and we have _NSIG of those entries, plus an ending \0 */
+ maxlen *= _NSIG+1;
+
+ *bufp = buf = kmalloc(maxlen, GFP_KERNEL);
+ if (!buf)
+  return -ENOMEM;
+
+ spin_lock(&sighand->siglock);
+ for (i=0; i<_NSIG; i++) {
```

```
+  action = &sighand->action[i];
+  buf += sprintf(buf, "%10d %20lu ", i, action->sa.sa_flags);
+  buf += print_sigset(buf, &action->sa.sa_mask);
+  buf += sprintf(buf, " %20lu\n", (unsigned long)action->sa.sa_handler);
+  BUG_ON(buf-*bufp > maxlen);
+ }
+ spin_unlock(&sighand->siglock);
+ return buf-*bufp;
+}
+
+int task_read_procsig(struct task_struct *p, char *name, char **value)
+{
+ int ret;
+
+ if (current != p) {
+  ret = security_ptrace(current, p);
+  if (ret)
+   return ret;
+ }
+
+ ret = -EINVAL;
+
+ if (strcmp(name, "pending") == 0) {
+  /* not masking out blocked signals  yet */
+  ret = print_sigpending_alloc(value, &p->pending);
+ } else if (strcmp(name, "blocked") == 0) {
+  /* not masking out blocked signals  yet */
+  ret = print_sigset_alloc(value, &p->blocked);
+ } else if (strcmp(name, "shared_pending") == 0) {
+  ret = print_sigpending_alloc(value, &p->signal->shared_pending);
+ } else if (strcmp(name, "waiters") == 0) {
+  ret = print_wait_info(value, p->signal);
+ } else if (strcmp(name, "action") == 0) {
+  ret = print_sigaction_list(value, p->sighand);
+ }  else if (strcmp(name, "altstack") == 0) {
+  *value = kmalloc(40, GFP_KERNEL);
+  ret = -ENOMEM;
+  if (*value) {
+   ret = sprintf(*value, "%lu %zd", p->sas_ss_sp, p->sas_ss_size);
+  }
+ }
+
+ return ret;
+}
--
1.5.1.1.GIT
```

Subject: [PATCH 2/2] signal c/r: implement /proc/pid/sig writing
Posted by serue on Fri, 08 Jun 2007 15:54:58 GMT
View Forum Message <> Reply to Message

>From 0fc34dcb54fe80ea720225825ad1dcb1b847d8ab Mon Sep 17 00:00:00 2001
From: Serge E. Hallyn <serue@us.ibm.com>
Date: Thu, 17 May 2007 17:28:07 -0400
Subject: [PATCH 2/2] signal c/r: implement /proc/pid/sig writing

Allow restore through writes to /proc/pid/sig/*, except for
the waiters file. Waits must be restored by the waiter actually
running wait.

To test writes to the action file I use the included handlertest.c.
Start it up in one terminal as
 ./handlertest 1
It should print "using handler 1"
in another terminal
 pid=`ps -ef | grep handlertest | grep -v grep | awk '{ print $2 '}`
 cat /proc/$pid/sig/action > action
 kill -USR1 $pid
handlertest should print "handler 1 called"

You can repeat this with
 ./handlertest 2
to make sure it does what you expect with the second signal handler.

Restart handlertest with a different signal handler:
 ./handlertest 2
It prints "using handler 2"
Overwrite it's sigactions from another terminal
 pid=`ps -ef | grep handlertest | grep -v grep | awk '{ print $2 '}`
 cat action > /proc/$pid/sig/action
and send it a USR1
 kill -USR1 $pid
and it should say "handler 1 called".

==========
handlertest.c
==========

```
void handler1(int sig)
{
```

```
 printf("handler 1 called\n");
}

void handler2(int sig)
{
 printf("handler 2 called\n");
}

int main(int argc, char *argv[])
{
 int hnum;
 __sighandler_t h = handler1;

 if (argc<2) {
  printf("usage: %s [1|2]\n", argv[0]);
  exit(2);
 }
 hnum = atoi(argv[1]);
 if (hnum == 1) {
  printf("using handler 1\n");
 } else {
  h = handler2;
  printf("using handler 2\n");
 }
 signal(SIGUSR1, h);

 sleep(100);
}

Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>
---
 fs/proc/base.c         |   97 ++++++++++++++++++++++-
 include/linux/signal.h |    2 +
 kernel/signal.c        |  218 ++++++++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 313 insertions(+), 4 deletions(-)

diff --git a/fs/proc/base.c b/fs/proc/base.c
index 8e0dd7a..22817bb 100644
--- a/fs/proc/base.c
+++ b/fs/proc/base.c
@@ -1701,9 +1701,104 @@ static ssize_t proc_pid_sig_read(struct file * file, char __user * buf,
  return length;
 }

+struct proc_pid_sig_partial {
+ char *buf;
+ loff_t pos;
+ size_t count;
```

```
+};
+
+static struct proc_pid_sig_partial *make_partial(char *page, int start, int total_count)
+{
+ struct proc_pid_sig_partial *partial;
+
+ partial = kzalloc(sizeof(*partial), GFP_KERNEL);
+ if (!partial)
+  return NULL;
+ partial->buf = page;
+ partial->pos = start;
+ partial->count = total_count;
+ return partial;
+}
+
+static ssize_t proc_pid_sig_write(struct file * file, const char __user * buf,
+       size_t count, loff_t *ppos)
+{
+ struct dentry * dentry = file->f_path.dentry;
+ struct inode * inode = dentry->d_inode;
+ char *page;
+ ssize_t length;
+ struct task_struct *task = get_proc_task(inode);
+ struct proc_pid_sig_partial *partial;
+ int partial_size = 0, total_count;
+
+ length = -ESRCH;
+ if (!task)
+  goto out_no_task;
+
+ partial = file->private_data;
+ file->private_data = NULL;
+ if (partial)
+  partial_size = partial->count - partial->pos;
+ total_count = count + partial_size;
+
+ printk(KERN_NOTICE "%s: 1\n", __FUNCTION__);
+ length = -ENOMEM;
+ page = kmalloc(total_count, GFP_USER);
+ if (!page)
+  goto out_put_task;
+ printk(KERN_NOTICE "%s: 2\n", __FUNCTION__);
+
+ if (partial) {
+  memcpy(page, partial->buf + partial->pos, partial_size);
+  kfree(partial->buf);
+  kfree(partial);
+ }
```

```
+
+ printk(KERN_NOTICE "%s: 3\n", __FUNCTION__);
+ length = -EFAULT;
+ if (copy_from_user(page+partial_size, buf, count))
+  goto out_free_page;
+
+ length = task_write_procsig(task,
+          (char*)file->f_path.dentry->d_name.name,
+          (void*)page, total_count);
+
+ printk(KERN_NOTICE "%s: 4, length was %d, count %d totcnt %d\n", __FUNCTION__,
+   length, count, total_count);
+ if (length >= 0 && length < total_count) {
+  /* should i just allocate a new buffer for just the unread portion? */
+  file->private_data = make_partial(page, length+1, total_count);
+  if (!file->private_data)
+   length = -ENOMEM;
+  else
+   goto out_put_task; /* don't free page, partial points to it */
+ }
+
+out_free_page:
+ kfree(page);
+out_put_task:
+ put_task_struct(task);
+out_no_task:
+ if (length >= 0)
+  return count;
+ return length;
+}
+
+static int proc_pid_sig_release(struct inode *inode, struct file *file)
+{
+ struct proc_pid_sig_partial *partial = file->private_data;
+
+ if (partial) {
+  kfree(partial->buf);
+  kfree(partial);
+ }
+ return 0;
+}
+
 static const struct file_operations proc_sig_attr_operations = {
   .read  = proc_pid_sig_read,
-// .write  = proc_pid_sig_write,
+ .write  = proc_pid_sig_write,
+ .release = proc_pid_sig_release,
 };
```

```
diff --git a/include/linux/signal.h b/include/linux/signal.h
index 6863543..2e6d64c 100644
--- a/include/linux/signal.h
+++ b/include/linux/signal.h
@@ -244,6 +244,8 @@ extern int get_signal_to_deliver(siginfo_t *info, struct k_sigaction
*return_ka,

 extern struct kmem_cache *sighand_cachep;
 extern int task_read_procsig(struct task_struct *p, char *name, char **value);
+extern int task_write_procsig(struct task_struct *p, char *name, void *value,
+   size_t size);

 /*
  * In POSIX a signal is sent either to a specific thread (Linux task)
diff --git a/kernel/signal.c b/kernel/signal.c
index a2a3ebe..dc50e1b 100644
--- a/kernel/signal.c
+++ b/kernel/signal.c
@@ -2614,7 +2614,7 @@ static int print_sigpending_alloc(char **bufp, struct sigpending
*pending)

  list_for_each_entry(q, &pending->list, list) {
   info = &q->info;
-  if (p-buf+215 > alloced) {
+  if (p-buf+221 > alloced) {
    int len=p-buf;
    char *buf2;
    alloced += PAGE_SIZE;
@@ -2629,8 +2629,8 @@ static int print_sigpending_alloc(char **bufp, struct sigpending
*pending)
    p = buf+len;
   }

-  p += sprintf(p, "sig %d: user %d flags %d",
-   info->si_signo, (int)q->user->uid, q->flags);
+  p += sprintf(p, "sig %d: uid %d pid %d flags %d",
+   info->si_signo, (int)q->user->uid, info->si_pid, q->flags);
   p += sprintf(p, " errno %d code %d\n",
    info->si_errno, info->si_code);

@@ -2739,3 +2739,215 @@ int task_read_procsig(struct task_struct *p, char *name, char
**value)

  return ret;
 }
+
```

```
+static int set_sigset(sigset_t *sig, char *value, int size)
+{
+ int i;
+
+ if (size < _NSIG)
+  return -EINVAL;
+ sigemptyset(sig);
+ for (i=0; i<_NSIG; i++)
+  if (value[i] == '1')
+   sigaddset(sig, i);
+ return 0;
+}
+
+static char *next_eol(char *s, int maxlen)
+{
+ int len=0;
+ while (len<maxlen && *s!='\0' && *s !='\n')
+  s++, len++;
+ if (len < maxlen)
+  return s;
+ return NULL;
+}
+
+static int set_sigpending(struct task_struct *t, struct sigpending *pending,
+     char *value, int size)
+{
+ struct sigqueue *q;
+ int ret;
+#ifdef __ARCH_SI_TRAPNO
+ int trapno;
+#endif
+
+ if (set_sigset(&pending->signal, value, size))
+  return -EINVAL;
+
+ size -= _NSIG+1;
+ value += _NSIG+1;
+ while (size) {
+  int signum, uid, pid, flags, errno, code, status, fd;
+  unsigned long utime, stime, addr, band;
+  char *start, *eol = next_eol(value, size);
+
+  if (!eol)
+   return 0;
+
+  size -= (eol - value) + 1;
+  start = eol+1;
+  ret = sscanf(value, "sig %d: uid %d pid %d flags %d errno %d code %d\n",
```

```
+   &signum, &uid, &pid, &flags, &errno, &code);
+  if (ret != 6)
+   goto out;
+  q = __sigqueue_alloc(t, GFP_KERNEL, 1);
+  if (!q)
+   goto out;
+  q->info.si_signo = signum;
+  q->info.si_errno = errno;
+  q->info.si_code = code;
+  q->info.si_uid = uid;
+  q->info.si_pid = pid;
+
+  switch(signum) {
+   /* XXX skipping posix1b timers and signals for now */
+   default: break;
+   case SIGKILL:
+    eol = next_eol(start, size);
+    if (!eol)
+     goto out;
+    size -= (eol - start) + 1;
+    ret = sscanf(start, " pid %d uid %d\n", &pid, &uid);
+    if (ret != 2)
+     goto out;
+    q->info._sifields._kill._pid = pid;
+    q->info._sifields._kill._uid = uid;
+    break;
+
+   case SIGCHLD:
+    eol = next_eol(start, size);
+    if (!eol)
+     goto out;
+    size -= (eol - start) + 1;
+    ret = sscanf(start, "pid %d uid %d status %d utime %lu stime %lu\n",
+     &pid, &uid, &status, &utime, &stime);
+    if (ret != 5)
+     goto out;
+    q->info._sifields._sigchld._pid = pid;
+    q->info._sifields._sigchld._uid = uid;
+    q->info._sifields._sigchld._status = status;
+    q->info._sifields._sigchld._utime = utime;
+    q->info._sifields._sigchld._stime = stime;
+    break;
+
+   case SIGILL:
+   case SIGFPE:
+   case SIGSEGV:
+   case SIGBUS:
+    eol = next_eol(start, size);
```

```
+    if (!eol)
+     goto out;
+    size -= (eol - start) + 1;
+#ifdef __ARCH_SI_TRAPNO
+    ret = sscanf(start, " addr %lu trapno %d\n", &addr, &trapno);
+    if (ret != 2)
+     goto out;
+    q->info._sifields._sigfault._trapno = trapno;
+#else
+    ret = sscanf(start, " addr %lu\n", &addr);
+    if (ret != 1)
+     goto out;
+#endif
+    q->info._sifields._sigfault._addr = (void __user *)addr;
+    break;
+
+   case SIGPOLL:
+    eol = next_eol(start, size);
+    if (!eol)
+     goto out;
+    size -= (eol - start) + 1;
+    ret = sscanf(start, " band %ld fd %d\n", &band, &fd);
+    if (ret != 2)
+     goto out;
+    q->info._sifields._sigpoll._band = band;
+    q->info._sifields._sigpoll._fd = fd;
+    break;
+  }
+  start = eol+1;
+  list_add_tail(&q->list, &pending->list);
+ }
+
+out:
+ return size;
+}
+
+static int set_sigaction_list(struct sighand_struct *sighand, char *value,
+    int size)
+{
+ struct k_sigaction *action;
+ char *eol;
+ int num, ret;
+ int origsize = size;
+ //char sa_mask[_NSIG+1];
+ char sa_mask[65];
+ unsigned long sa_flags, sa_handler;
+
+ while (size > 0 && (eol=next_eol(value, size))) {
```

```
+  sa_mask[64] = '\0';
+  ret = sscanf(value, "%d %lu %64s %lu\n",
+   &num, &sa_flags, sa_mask, &sa_handler);
+  if (ret != 4)
+   break;
+  if (num < 0 || num > _NSIG)
+   return -EINVAL;
+  action = &sighand->action[num];
+  action->sa.sa_flags = sa_flags;
+  set_sigset(&action->sa.sa_mask, sa_mask, _NSIG);
+  action->sa.sa_handler = (__sighandler_t) sa_handler;
+  size -= eol-value+1;
+  value = eol+1;
+ }
+
+ return origsize - size;
+}
+
+static int set_altstack(struct task_struct *p, char *value, int size)
+{
+ unsigned long sas_ss_sp;
+ size_t sas_ss_size;
+ int ret;
+
+ ret = sscanf(value, "%lu %zd\n", &sas_ss_sp, &sas_ss_size);
+ if (ret != 2)
+  return -EINVAL;
+ p->sas_ss_sp = sas_ss_sp;
+ p->sas_ss_size = sas_ss_size;
+
+ return size;
+}
+
+int task_write_procsig(struct task_struct *p, char *name, void *value,
+   size_t size)
+{
+ int ret;
+
+ if (current != p) {
+  ret = security_ptrace(current, p);
+  if (ret)
+   return ret;
+ }
+
+ ret = -EINVAL;
+
+ if (strcmp(name, "pending") == 0) {
+  /* not masking out blocked signals  yet */
```

```
+  ret = set_sigpending(p, &p->pending, value, size);
+ } else if (strcmp(name, "shared_pending") == 0) {
+  ret = set_sigpending(p, &p->signal->shared_pending, value, size);
+ } else if (strcmp(name, "blocked") == 0) {
+  /* not masking out blocked signals  yet */
+  ret = set_sigset(&p->blocked, value, size);
+ } else if (strcmp(name, "action") == 0) {
+  ret = set_sigaction_list(p->sighand, value, size);
+ }  else if (strcmp(name, "altstack") == 0) {
+  ret = set_altstack(p, value, size);
+#if 0
+ } else if (strcmp(name, "waiters") == 0) {
+  /* reset this at the waiter's restart using do_wait */
+  ret = set_wait_info(p->signal, value,  size);
+#endif
+ }
+
+ return ret;
+}
--
1.5.1.1.GIT
```

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by Cedric Le Goater on Mon, 11 Jun 2007 14:43:12 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> As I mentioned earlier, I don't know what sort of approach we want
> to take to guide checkpoint and restart. I.e. do we want it to be
> a mostly userspace-orchestrated affair, or entirely done in the
> kernel using the freezer or some other mechanism in response to a
> single syscall or containerfs file write?
>
> If we wanted to do a lot of the work in userspace, here is a pair of
> patches to read and restore signal information. It's entirely unsafe
> wrt locking, bc i would assume that if we did in fact do c/r from
> userspace, we would have some way of entirely pulling the task off
> the runqueue while doing our thing...
>
> Anyway, this is purely to start discussion.
>
> thanks,

> -serge
>
>>From 30bbe322942e5ed86bad63861dad80595cd04063 Mon Sep 17 00:00:00 2001
> From: Serge E. Hallyn <serue@us.ibm.com>
> Date: Mon, 30 Apr 2007 16:22:44 -0400
> Subject: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
>
> Define /proc/<pid>/sig/ directory containing files to report
> on a process' signal info.
>
> Files defined:
> action: list signal action
> altstack: print sigaltstack location and size
> blocked: print blocked signal mask
> pending: print pending signals and siginfo
> shared_pending: print shared pending signals and siginfo
> waiters: list tasks wait4()ing on task PID.

some of these are already in /proc/<pid>/status and /proc/<pid>/stat

should we continue to use /proc ? or switch to some other mechanisms
like getnetlink (taskstats) to map kernel structures.

[...]

> +/*
> + * print a sigset_t to a buffer. Return # characters printed,
> + * not including the final ending '\0'.
> + */
> +static int print_sigset(char *buf, sigset_t *sig)
> +{
> + int i;
> +
> + for (i=0; i<_NSIG; i++) {
> + if (sigismember(sig, i))
> +   buf[i] = '1';
> + else
> +   buf[i] = '0';
> + }
> + buf[_NSIG] = '\0';
> +
> + return _NSIG;
> +}

you might want to use render_sigset_t() in fs/proc array.C

> +static int print_sigset_alloc(char **bufp, sigset_t *sig)
> +{

```
> + char *buf;
> +
> + *bufp = buf = kmalloc(_NSIG+1, GFP_KERNEL);
> + if (!buf)
> +  return -ENOMEM;
> +
> + return print_sigset(buf, sig);
> +}
> +
> +static int print_wait_info(char **bufp, struct signal_struct *signal)
> +{
> + char *buf;
> + wait_queue_t *wait;
> +
> + *bufp = buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
> + if (!buf)
> +  return -ENOMEM;
> +
> + spin_lock(&signal->wait_chldexit.lock);
> +
> + list_for_each_entry(wait, &signal->wait_chldexit.task_list, task_list) {
> +  struct task_struct *tsk = wait->private;
> +
> +  if (buf - *bufp +50 > PAGE_SIZE) {
> +   spin_unlock(&signal->wait_chldexit.lock);
> +   kfree(buf);
> +   *bufp = NULL;
> +   return -ENOMEM;
> +  }
> +  WARN_ON(wait->func != default_wake_function);
> +  buf += sprintf(buf, "%u %d\n", wait->flags, tsk->pid);
> + }
> +
> + spin_unlock(&signal->wait_chldexit.lock);
> +
> + return buf-*bufp;
> +}
> +
> +static int print_sigpending_alloc(char **bufp, struct sigpending *pending)
> +{
> + int alloced=0;
> + char *buf, *p;
> + struct sigqueue *q;
> + struct siginfo *info;
> +
> + alloced = PAGE_SIZE;
> + p = buf = kmalloc(alloced, GFP_KERNEL);
> + if (!buf)
```

```
> +  return -ENOMEM;
> +
> + p += print_sigset(buf, &pending->signal);
> + p += sprintf(p, "\n");
> +
> + list_for_each_entry(q, &pending->list, list) {
> +  info = &q->info;
> +  if (p-buf+215 > alloced) {
> +   int len=p-buf;
> +   char *buf2;
> +   alloced += PAGE_SIZE;
> +   buf2 = kmalloc(alloced, GFP_KERNEL);
> +   if (!buf2) {
> +    kfree(buf);
> +    return -ENOMEM;
> +   }
> +   memcpy(buf2, buf, alloced - PAGE_SIZE);
> +   kfree(buf);
> +   buf = buf2;
> +   p = buf+len;
> +  }
> +
> +  p += sprintf(p, "sig %d: user %d flags %d",
> +   info->si_signo, (int)q->user->uid, q->flags);
> +  p += sprintf(p, " errno %d code %d\n",
> +   info->si_errno, info->si_code);
> +
> +  switch(info->si_signo) {
> +  case SIGKILL:
> +   p += sprintf(p, "  spid %d suid %d\n",
> +    info->_sifields._kill._pid,
> +    info->_sifields._kill._uid);
> +   break;
> +  /* XXX skipping posix1b timers and signals for now */
> +  case SIGCHLD:
> +   p += sprintf(p, "  pid %d uid %d status %d utime %lu stime %lu\n",
> +    info->_sifields._sigchld._pid,
> +    info->_sifields._sigchld._uid,
> +    info->_sifields._sigchld._status,
> +    info->_sifields._sigchld._utime,
> +    info->_sifields._sigchld._stime);
> +   break;
> +  case SIGILL:
> +  case SIGFPE:
> +  case SIGSEGV:
> +  case SIGBUS:
> +#ifdef __ARCH_SI_TRAPNO
> +   p += sprintf(p, "  addr %lu trapno %d\n",
```

```
> +    (unsigned long)info->_sifields._sigfault._addr,
> +    info->_sifields._sigfault._trapno);
> +#else
> +   p += sprintf(p, "  addr %lu\n",
> +    (unsigned long)info->_sifields._sigfault._addr);
> +#endif
> +   break;
> +  case SIGPOLL:
> +   p += sprintf(p, "  band %ld fd %d\n",
> +    (long)info->_sifields._sigpoll._band,
> +    info->_sifields._sigpoll._fd);
> +   break;
> +  default:
> +   p += sprintf(p, "  Unsupported siginfo for signal %d\n",
> +    info->si_signo);
> +   break;
> +  }
> + }
> + *bufp = buf;
> + return p-buf;
> +}
```

I think we are reaching the limit of /proc when we expose the pending siginfos.

```
> +static int print_sigaction_list(char **bufp, struct sighand_struct *sighand)
> +{
> + struct k_sigaction *action;
> + int maxlen;
> + int i;
> + char *buf;
> +
> + /* two unsigned longs (20 chars), one int (10 chars), a sigset_t, 3 spaces, plus a newline */
> + maxlen = 10 + 20*2 + _NSIG + 4;
> + /* and we have _NSIG of those entries, plus an ending \0 */
> + maxlen *= _NSIG+1;
> +
> + *bufp = buf = kmalloc(maxlen, GFP_KERNEL);
> + if (!buf)
> +  return -ENOMEM;
> +
> + spin_lock(&sighand->siglock);
> + for (i=0; i<_NSIG; i++) {
> +  action = &sighand->action[i];
> +  buf += sprintf(buf, "%10d %20lu ", i, action->sa.sa_flags);
> +  buf += print_sigset(buf, &action->sa.sa_mask);
> +  buf += sprintf(buf, " %20lu\n", (unsigned long)action->sa.sa_handler);
> +  BUG_ON(buf-*bufp > maxlen);
> + }
```

```
> + spin_unlock(&sighand->siglock);
> + return buf-*bufp;
> +}
> +
> +int task_read_procsig(struct task_struct *p, char *name, char **value)
> +{
> + int ret;
> +
> + if (current != p) {
> +  ret = security_ptrace(current, p);
> +  if (ret)
> +   return ret;
> + }
> +
> + ret = -EINVAL;
> +
> + if (strcmp(name, "pending") == 0) {
> +  /* not masking out blocked signals  yet */
> +  ret = print_sigpending_alloc(value, &p->pending);
> + } else if (strcmp(name, "blocked") == 0) {
> +  /* not masking out blocked signals  yet */
> +  ret = print_sigset_alloc(value, &p->blocked);
> + } else if (strcmp(name, "shared_pending") == 0) {
> +  ret = print_sigpending_alloc(value, &p->signal->shared_pending);
> + } else if (strcmp(name, "waiters") == 0) {
> +  ret = print_wait_info(value, p->signal);
> + } else if (strcmp(name, "action") == 0) {
> +  ret = print_sigaction_list(value, p->sighand);
> + }  else if (strcmp(name, "altstack") == 0) {
> +  *value = kmalloc(40, GFP_KERNEL);
> +  ret = -ENOMEM;
> +  if (*value) {
> +   ret = sprintf(*value, "%lu %zd", p->sas_ss_sp, p->sas_ss_size);
> +  }
> + }
> +
> + return ret;
> +}
```

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 2/2] signal c/r: implement /proc/pid/sig writing
Posted by Cedric Le Goater on Mon, 11 Jun 2007 14:53:16 GMT

with all the infos you've gathered in /proc, why don't you just kill the
process ?

The patch we have to restore pending signals in 2.6.21-mm2-lxc3 does :

```
+static int pid_set_siginfo(mcrk_session_t * s, void *ptarg)
+{
+ mcrk_pid_setsignal_t arg;
+ siginfo_t si;
+ int ret;
+
+ if (!ptarg) {
+  return -EINVAL;
+ }
+
+ if (copy_from_user(&arg, ptarg, sizeof(arg)))
+  return -EFAULT;
+ if (copy_from_user(&si, U64_2_PTR(arg.siginfo), sizeof(si)))
+  return -EFAULT;
+
+ if (arg.shared) {
+  ret = kill_proc_info(si.si_signo, &si, current->pid);
+ } else {
+  ret = send_sig_info(si.si_signo, &si, current);
+ }
+ return ret;
+}
```

C.

Serge E. Hallyn wrote:
>>From 0fc34dcb54fe80ea720225825ad1dcb1b847d8ab Mon Sep 17 00:00:00 2001
> From: Serge E. Hallyn <serue@us.ibm.com>
> Date: Thu, 17 May 2007 17:28:07 -0400
> Subject: [PATCH 2/2] signal c/r: implement /proc/pid/sig writing
>
> Allow restore through writes to /proc/pid/sig/*, except for
> the waiters file. Waits must be restored by the waiter actually
> running wait.
>
> To test writes to the action file I use the included handlertest.c.
> Start it up in one terminal as
> ./handlertest 1
> It should print "using handler 1"
> in another terminal
> pid=`ps -ef | grep handlertest | grep -v grep | awk '{ print $2 '}`

> cat /proc/$pid/sig/action > action
> kill -USR1 $pid
> handlertest should print "handler 1 called"
>
> You can repeat this with
> ./handlertest 2
> to make sure it does what you expect with the second signal handler.
>
> Restart handlertest with a different signal handler:
> ./handlertest 2
> It prints "using handler 2"
> Overwrite it's sigactions from another terminal
> pid=`ps -ef | grep handlertest | grep -v grep | awk '{ print $2 '}`
> cat action > /proc/$pid/sig/action
> and send it a USR1
> kill -USR1 $pid
> and it should say "handler 1 called".
>
> ==========
> handlertest.c
> ==========
>
> void handler1(int sig)
> {
> printf("handler 1 called\n");
> }
>
> void handler2(int sig)
> {
> printf("handler 2 called\n");
> }
>
> int main(int argc, char *argv[])
> {
> int hnum;
> __sighandler_t h = handler1;
>
> if (argc<2) {
>   printf("usage: %s [1|2]\n", argv[0]);
>   exit(2);
> }
> hnum = atoi(argv[1]);
> if (hnum == 1) {
>   printf("using handler 1\n");
> } else {
>   h = handler2;
>   printf("using handler 2\n");
> }

> signal(SIGUSR1, h);
>
> sleep(100);
> }
>
> Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>
> ---
> fs/proc/base.c | 97 ++++++++++++++++++++++++-
> include/linux/signal.h | 2 +
> kernel/signal.c | 218 ++-
> 3 files changed, 313 insertions(+), 4 deletions(-)
>
> diff --git a/fs/proc/base.c b/fs/proc/base.c
> index 8e0dd7a..22817bb 100644
> --- a/fs/proc/base.c
> +++ b/fs/proc/base.c
> @@ -1701,9 +1701,104 @@ static ssize_t proc_pid_sig_read(struct file * file, char __user * buf,
>   return length;
> }
>
> +struct proc_pid_sig_partial {
> + char *buf;
> + loff_t pos;
> + size_t count;
> +};
> +
> +static struct proc_pid_sig_partial *make_partial(char *page, int start, int total_count)
> +{
> + struct proc_pid_sig_partial *partial;
> +
> + partial = kzalloc(sizeof(*partial), GFP_KERNEL);
> + if (!partial)
> + return NULL;
> + partial->buf = page;
> + partial->pos = start;
> + partial->count = total_count;
> + return partial;
> +}
> +
> +static ssize_t proc_pid_sig_write(struct file * file, const char __user * buf,
> +   size_t count, loff_t *ppos)
> +{
> + struct dentry * dentry = file->f_path.dentry;
> + struct inode * inode = dentry->d_inode;
> + char *page;
> + ssize_t length;
> + struct task_struct *task = get_proc_task(inode);
> + struct proc_pid_sig_partial *partial;

```
> + int partial_size = 0, total_count;
> +
> + length = -ESRCH;
> + if (!task)
> +  goto out_no_task;
> +
> + partial = file->private_data;
> + file->private_data = NULL;
> + if (partial)
> +  partial_size = partial->count - partial->pos;
> + total_count = count + partial_size;
> +
> + printk(KERN_NOTICE "%s: 1\n", __FUNCTION__);
> + length = -ENOMEM;
> + page = kmalloc(total_count, GFP_USER);
> + if (!page)
> +  goto out_put_task;
> + printk(KERN_NOTICE "%s: 2\n", __FUNCTION__);
> +
> + if (partial) {
> +  memcpy(page, partial->buf + partial->pos, partial_size);
> +  kfree(partial->buf);
> +  kfree(partial);
> + }
> +
> + printk(KERN_NOTICE "%s: 3\n", __FUNCTION__);
> + length = -EFAULT;
> + if (copy_from_user(page+partial_size, buf, count))
> +  goto out_free_page;
> +
> + length = task_write_procsig(task,
> +         (char*)file->f_path.dentry->d_name.name,
> +         (void*)page, total_count);
> +
> + printk(KERN_NOTICE "%s: 4, length was %d, count %d totcnt %d\n", __FUNCTION__,
> +   length, count, total_count);
> + if (length >= 0 && length < total_count) {
> +  /* should i just allocate a new buffer for just the unread portion? */
> +  file->private_data = make_partial(page, length+1, total_count);
> +  if (!file->private_data)
> +   length = -ENOMEM;
> +  else
> +   goto out_put_task; /* don't free page, partial points to it */
> + }
> +
> +out_free_page:
> + kfree(page);
> +out_put_task:
```

> + put_task_struct(task);
> +out_no_task:
> + if (length >= 0)
> + return count;
> + return length;
> +}
> +
> +static int proc_pid_sig_release(struct inode *inode, struct file *file)
> +{
> + struct proc_pid_sig_partial *partial = file->private_data;
> +
> + if (partial) {
> + kfree(partial->buf);
> + kfree(partial);
> + }
> + return 0;
> +}
> +
> static const struct file_operations proc_sig_attr_operations = {
>   .read = proc_pid_sig_read,
> -// .write = proc_pid_sig_write,
> + .write = proc_pid_sig_write,
> + .release = proc_pid_sig_release,
> };
>
>
> diff --git a/include/linux/signal.h b/include/linux/signal.h
> index 6863543..2e6d64c 100644
> --- a/include/linux/signal.h
> +++ b/include/linux/signal.h
> @@ -244,6 +244,8 @@ extern int get_signal_to_deliver(siginfo_t *info, struct k_sigaction
*return_ka,
>
> extern struct kmem_cache *sighand_cachep;
> extern int task_read_procsig(struct task_struct *p, char *name, char **value);
> +extern int task_write_procsig(struct task_struct *p, char *name, void *value,
> +   size_t size);
>
> /*
>   * In POSIX a signal is sent either to a specific thread (Linux task)
> diff --git a/kernel/signal.c b/kernel/signal.c
> index a2a3ebe..dc50e1b 100644
> --- a/kernel/signal.c
> +++ b/kernel/signal.c
> @@ -2614,7 +2614,7 @@ static int print_sigpending_alloc(char **bufp, struct sigpending
*pending)
>
>   list_for_each_entry(q, &pending->list, list) {

```
>    info = &q->info;
> -  if (p-buf+215 > alloced) {
> +  if (p-buf+221 > alloced) {
>     int len=p-buf;
>     char *buf2;
>     alloced += PAGE_SIZE;
> @@ -2629,8 +2629,8 @@ static int print_sigpending_alloc(char **bufp, struct sigpending
*pending)
>     p = buf+len;
>    }
>
> -  p += sprintf(p, "sig %d: user %d flags %d",
> -   info->si_signo, (int)q->user->uid, q->flags);
> +  p += sprintf(p, "sig %d: uid %d pid %d flags %d",
> +   info->si_signo, (int)q->user->uid, info->si_pid, q->flags);
>    p += sprintf(p, " errno %d code %d\n",
>     info->si_errno, info->si_code);
>
> @@ -2739,3 +2739,215 @@ int task_read_procsig(struct task_struct *p, char *name, char
**value)
>
>   return ret;
>  }
> +
> +static int set_sigset(sigset_t *sig, char *value, int size)
> +{
> + int i;
> +
> + if (size < _NSIG)
> +  return -EINVAL;
> + sigemptyset(sig);
> + for (i=0; i<_NSIG; i++)
> +  if (value[i] == '1')
> +   sigaddset(sig, i);
> + return 0;
> +}
> +
> +static char *next_eol(char *s, int maxlen)
> +{
> + int len=0;
> + while (len<maxlen && *s!='\0' && *s !='\n')
> +  s++, len++;
> + if (len < maxlen)
> +  return s;
> + return NULL;
> +}
> +
> +static int set_sigpending(struct task_struct *t, struct sigpending *pending,
```

```
> +     char *value, int size)
> +{
> + struct sigqueue *q;
> + int ret;
> +#ifdef __ARCH_SI_TRAPNO
> + int trapno;
> +#endif
> +
> + if (set_sigset(&pending->signal, value, size))
> +  return -EINVAL;
> +
> + size -= _NSIG+1;
> + value += _NSIG+1;
> + while (size) {
> +  int signum, uid, pid, flags, errno, code, status, fd;
> +  unsigned long utime, stime, addr, band;
> +  char *start, *eol = next_eol(value, size);
> +
> +  if (!eol)
> +   return 0;
> +
> +  size -= (eol - value) + 1;
> +  start = eol+1;
> +  ret = sscanf(value, "sig %d: uid %d pid %d flags %d errno %d code %d\n",
> +   &signum, &uid, &pid, &flags, &errno, &code);
> +  if (ret != 6)
> +   goto out;
> +  q = __sigqueue_alloc(t, GFP_KERNEL, 1);
> +  if (!q)
> +   goto out;
> +  q->info.si_signo = signum;
> +  q->info.si_errno = errno;
> +  q->info.si_code = code;
> +  q->info.si_uid = uid;
> +  q->info.si_pid = pid;
> +
> +  switch(signum) {
> +   /* XXX skipping posix1b timers and signals for now */
> +   default: break;
> +   case SIGKILL:
> +    eol = next_eol(start, size);
> +    if (!eol)
> +     goto out;
> +    size -= (eol - start) + 1;
> +    ret = sscanf(start, " pid %d uid %d\n", &pid, &uid);
> +    if (ret != 2)
> +     goto out;
> +    q->info._sifields._kill._pid = pid;
```

```
> +    q->info._sifields._kill._uid = uid;
> +    break;
> +
> +   case SIGCHLD:
> +    eol = next_eol(start, size);
> +    if (!eol)
> +     goto out;
> +    size -= (eol - start) + 1;
> +    ret = sscanf(start, "pid %d uid %d status %d utime %lu stime %lu\n",
> +     &pid, &uid, &status, &utime, &stime);
> +    if (ret != 5)
> +     goto out;
> +    q->info._sifields._sigchld._pid = pid;
> +    q->info._sifields._sigchld._uid = uid;
> +    q->info._sifields._sigchld._status = status;
> +    q->info._sifields._sigchld._utime = utime;
> +    q->info._sifields._sigchld._stime = stime;
> +    break;
> +
> +   case SIGILL:
> +   case SIGFPE:
> +   case SIGSEGV:
> +   case SIGBUS:
> +    eol = next_eol(start, size);
> +    if (!eol)
> +     goto out;
> +    size -= (eol - start) + 1;
> +#ifdef __ARCH_SI_TRAPNO
> +    ret = sscanf(start, " addr %lu trapno %d\n", &addr, &trapno);
> +    if (ret != 2)
> +     goto out;
> +    q->info._sifields._sigfault._trapno = trapno;
> +#else
> +    ret = sscanf(start, " addr %lu\n", &addr);
> +    if (ret != 1)
> +     goto out;
> +#endif
> +    q->info._sifields._sigfault._addr = (void __user *)addr;
> +    break;
> +
> +   case SIGPOLL:
> +    eol = next_eol(start, size);
> +    if (!eol)
> +     goto out;
> +    size -= (eol - start) + 1;
> +    ret = sscanf(start, " band %ld fd %d\n", &band, &fd);
> +    if (ret != 2)
> +     goto out;
```

```
> +    q->info._sifields._sigpoll._band = band;
> +    q->info._sifields._sigpoll._fd = fd;
> +    break;
> +  }
> +  start = eol+1;
> +  list_add_tail(&q->list, &pending->list);
> + }
> +
> +out:
> + return size;
> +}
> +
> +static int set_sigaction_list(struct sighand_struct *sighand, char *value,
> +    int size)
> +{
> + struct k_sigaction *action;
> + char *eol;
> + int num, ret;
> + int origsize = size;
> + //char sa_mask[_NSIG+1];
> + char sa_mask[65];
> + unsigned long sa_flags, sa_handler;
> +
> + while (size > 0 && (eol=next_eol(value, size))) {
> +  sa_mask[64] = '\0';
> +  ret = sscanf(value, "%d %lu %64s %lu\n",
> +   &num, &sa_flags, sa_mask, &sa_handler);
> +  if (ret != 4)
> +   break;
> +  if (num < 0 || num > _NSIG)
> +   return -EINVAL;
> +  action = &sighand->action[num];
> +  action->sa.sa_flags = sa_flags;
> +  set_sigset(&action->sa.sa_mask, sa_mask, _NSIG);
> +  action->sa.sa_handler = (__sighandler_t) sa_handler;
> +  size -= eol-value+1;
> +  value = eol+1;
> + }
> +
> + return origsize - size;
> +}
> +
> +static int set_altstack(struct task_struct *p, char *value, int size)
> +{
> + unsigned long sas_ss_sp;
> + size_t sas_ss_size;
> + int ret;
> +
```

```
> + ret = sscanf(value, "%lu %zd\n", &sas_ss_sp, &sas_ss_size);
> + if (ret != 2)
> +  return -EINVAL;
> + p->sas_ss_sp = sas_ss_sp;
> + p->sas_ss_size = sas_ss_size;
> +
> + return size;
> +}
> +
> +int task_write_procsig(struct task_struct *p, char *name, void *value,
> +   size_t size)
> +{
> + int ret;
> +
> + if (current != p) {
> +  ret = security_ptrace(current, p);
> +  if (ret)
> +   return ret;
> + }
> +
> + ret = -EINVAL;
> +
> + if (strcmp(name, "pending") == 0) {
> +  /* not masking out blocked signals  yet */
> +  ret = set_sigpending(p, &p->pending, value, size);
> + } else if (strcmp(name, "shared_pending") == 0) {
> +  ret = set_sigpending(p, &p->signal->shared_pending, value, size);
> + } else if (strcmp(name, "blocked") == 0) {
> +  /* not masking out blocked signals  yet */
> +  ret = set_sigset(&p->blocked, value, size);
> + } else if (strcmp(name, "action") == 0) {
> +  ret = set_sigaction_list(p->sighand, value, size);
> + }  else if (strcmp(name, "altstack") == 0) {
> +  ret = set_altstack(p, value, size);
> +#if 0
> + } else if (strcmp(name, "waiters") == 0) {
> +  /* reset this at the waiter's restart using do_wait */
> +  ret = set_wait_info(p->signal, value,  size);
> +#endif
> + }
> +
> + return ret;
> +}
```

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 2/2] signal c/r: implement /proc/pid/sig writing
Posted by serue on Mon, 11 Jun 2007 16:47:47 GMT
View Forum Message <> Reply to Message

Quoting Cedric Le Goater (clg@fr.ibm.com):
> with all the infos you've gathered in /proc, why don't you just kill the
> process ?
>
> The patch we have to restore pending signals in 2.6.21-mm2-lxc3 does :
>
> +static int pid_set_siginfo(mcrk_session_t * s, void *ptarg)
> +{
> + mcrk_pid_setsignal_t arg;
> + siginfo_t si;
> + int ret;
> +
> + if (!ptarg) {
> + return -EINVAL;
> + }
> +
> + if (copy_from_user(&arg, ptarg, sizeof(arg)))
> + return -EFAULT;
> + if (copy_from_user(&si, U64_2_PTR(arg.siginfo), sizeof(si)))
> + return -EFAULT;

Hmm, one problem with especially this second copy_from_user() is that
you are making the checkpoint image more kernel dependant.

Whatever approach we take both high-level and low-level, we do want to
avoid having checkpoint images directly reflect in-kernel structures,
right?

That's one area where the /proc approach has an inherent advantage over
using netlink to dump information, it avoids the temptation to just dump
and restore straight from the kernel pointer, which would threaten to
make restoring a checkpoint from another kernel much more dangerous.

> + if (arg.shared) {
> + ret = kill_proc_info(si.si_signo, &si, current->pid);
> + } else {
> + ret = send_sig_info(si.si_signo, &si, current);
> + }
> + return ret;
> +}

This part is fine with me, but assumes we take the more kernel-guided
approach, right.

And that's what I'm trying to get people to discuss :) Do we want a

more kernel-guided approach, or do we want to provide pieces of
functionality that userspace exploits?

Oh, or are you saying this would just replace the biggest chunk of my
set_sigpending() function below?

thanks,
-serge

>
> C.
>
> Serge E. Hallyn wrote:
> >>From 0fc34dcb54fe80ea720225825ad1dcb1b847d8ab Mon Sep 17 00:00:00 2001
> > From: Serge E. Hallyn <serue@us.ibm.com>
> > Date: Thu, 17 May 2007 17:28:07 -0400
> > Subject: [PATCH 2/2] signal c/r: implement /proc/pid/sig writing
> >
> > Allow restore through writes to /proc/pid/sig/*, except for
> > the waiters file. Waits must be restored by the waiter actually
> > running wait.
> >
> > To test writes to the action file I use the included handlertest.c.
> > Start it up in one terminal as
> > ./handlertest 1
> > It should print "using handler 1"
> > in another terminal
> > pid=`ps -ef | grep handlertest | grep -v grep | awk '{ print $2 '}`
> > cat /proc/$pid/sig/action > action
> > kill -USR1 $pid
> > handlertest should print "handler 1 called"
> >
> > You can repeat this with
> > ./handlertest 2
> > to make sure it does what you expect with the second signal handler.
> >
> > Restart handlertest with a different signal handler:
> > ./handlertest 2
> > It prints "using handler 2"
> > Overwrite it's sigactions from another terminal
> > pid=`ps -ef | grep handlertest | grep -v grep | awk '{ print $2 '}`
> > cat action > /proc/$pid/sig/action
> > and send it a USR1
> > kill -USR1 $pid
> > and it should say "handler 1 called".
> >
> > ==
> > handlertest.c

```
> > ===================================================================
> >
> > void handler1(int sig)
> > {
> >  printf("handler 1 called\n");
> > }
> >
> > void handler2(int sig)
> > {
> >  printf("handler 2 called\n");
> > }
> >
> > int main(int argc, char *argv[])
> > {
> >  int hnum;
> >  __sighandler_t h = handler1;
> >
> >  if (argc<2) {
> >   printf("usage: %s [1|2]\n", argv[0]);
> >   exit(2);
> >  }
> >  hnum = atoi(argv[1]);
> >  if (hnum == 1) {
> >   printf("using handler 1\n");
> >  } else {
> >   h = handler2;
> >   printf("using handler 2\n");
> >  }
> >  signal(SIGUSR1, h);
> >
> >  sleep(100);
> > }
> >
> > Signed-off-by: Serge E. Hallyn <serue@us.ibm.com>
> > ---
> >  fs/proc/base.c         |   97 ++++++++++++++++++++++++-
> >  include/linux/signal.h |    2 +
> >  kernel/signal.c        |  218 ++++++++++++++++++++++++++++++++++++++++++++++++-
> >  3 files changed, 313 insertions(+), 4 deletions(-)
> >
> > diff --git a/fs/proc/base.c b/fs/proc/base.c
> > index 8e0dd7a..22817bb 100644
> > --- a/fs/proc/base.c
> > +++ b/fs/proc/base.c
> > @@ -1701,9 +1701,104 @@ static ssize_t proc_pid_sig_read(struct file * file, char __user *
buf,
> >   return length;
> >  }
```

```
> >
> > +struct proc_pid_sig_partial {
> > + char *buf;
> > + loff_t pos;
> > + size_t count;
> > +};
> > +
> > +static struct proc_pid_sig_partial *make_partial(char *page, int start, int total_count)
> > +{
> > + struct proc_pid_sig_partial *partial;
> > +
> > + partial = kzalloc(sizeof(*partial), GFP_KERNEL);
> > + if (!partial)
> > +  return NULL;
> > + partial->buf = page;
> > + partial->pos = start;
> > + partial->count = total_count;
> > + return partial;
> > +}
> > +
> > +static ssize_t proc_pid_sig_write(struct file * file, const char __user * buf,
> > +       size_t count, loff_t *ppos)
> > +{
> > + struct dentry * dentry = file->f_path.dentry;
> > + struct inode * inode = dentry->d_inode;
> > + char *page;
> > + ssize_t length;
> > + struct task_struct *task = get_proc_task(inode);
> > + struct proc_pid_sig_partial *partial;
> > + int partial_size = 0, total_count;
> > +
> > + length = -ESRCH;
> > + if (!task)
> > +  goto out_no_task;
> > +
> > + partial = file->private_data;
> > + file->private_data = NULL;
> > + if (partial)
> > +  partial_size = partial->count - partial->pos;
> > + total_count = count + partial_size;
> > +
> > + printk(KERN_NOTICE "%s: 1\n", __FUNCTION__);
> > + length = -ENOMEM;
> > + page = kmalloc(total_count, GFP_USER);
> > + if (!page)
> > +  goto out_put_task;
> > + printk(KERN_NOTICE "%s: 2\n", __FUNCTION__);
> > +
```

```
> > + if (partial) {
> > +  memcpy(page, partial->buf + partial->pos, partial_size);
> > +  kfree(partial->buf);
> > +  kfree(partial);
> > + }
> > +
> > + printk(KERN_NOTICE "%s: 3\n", __FUNCTION__);
> > + length = -EFAULT;
> > + if (copy_from_user(page+partial_size, buf, count))
> > +  goto out_free_page;
> > +
> > + length = task_write_procsig(task,
> > +          (char*)file->f_path.dentry->d_name.name,
> > +          (void*)page, total_count);
> > +
> > + printk(KERN_NOTICE "%s: 4, length was %d, count %d totcnt %d\n", __FUNCTION__,
> > +   length, count, total_count);
> > + if (length >= 0 && length < total_count) {
> > +  /* should i just allocate a new buffer for just the unread portion? */
> > +  file->private_data = make_partial(page, length+1, total_count);
> > +  if (!file->private_data)
> > +   length = -ENOMEM;
> > +  else
> > +   goto out_put_task; /* don't free page, partial points to it */
> > + }
> > +
> > +out_free_page:
> > + kfree(page);
> > +out_put_task:
> > + put_task_struct(task);
> > +out_no_task:
> > + if (length >= 0)
> > +  return count;
> > + return length;
> > +}
> > +
> > +static int proc_pid_sig_release(struct inode *inode, struct file *file)
> > +{
> > + struct proc_pid_sig_partial *partial = file->private_data;
> > +
> > + if (partial) {
> > +  kfree(partial->buf);
> > +  kfree(partial);
> > + }
> > + return 0;
> > +}
> > +
> >  static const struct file_operations proc_sig_attr_operations = {
```

> >   .read = proc_pid_sig_read,
> > -// .write = proc_pid_sig_write,
> > + .write = proc_pid_sig_write,
> > + .release = proc_pid_sig_release,
> > };
> >
> >
> > diff --git a/include/linux/signal.h b/include/linux/signal.h
> > index 6863543..2e6d64c 100644
> > --- a/include/linux/signal.h
> > +++ b/include/linux/signal.h
> > @@ -244,6 +244,8 @@ extern int get_signal_to_deliver(siginfo_t *info, struct k_sigaction
*return_ka,
> >
> > extern struct kmem_cache *sighand_cachep;
> > extern int task_read_procsig(struct task_struct *p, char *name, char **value);
> > +extern int task_write_procsig(struct task_struct *p, char *name, void *value,
> > +   size_t size);
> >
> > /*
> >   * In POSIX a signal is sent either to a specific thread (Linux task)
> > diff --git a/kernel/signal.c b/kernel/signal.c
> > index a2a3ebe..dc50e1b 100644
> > --- a/kernel/signal.c
> > +++ b/kernel/signal.c
> > @@ -2614,7 +2614,7 @@ static int print_sigpending_alloc(char **bufp, struct sigpending
*pending)
> >
> >   list_for_each_entry(q, &pending->list, list) {
> >   info = &q->info;
> > - if (p-buf+215 > alloced) {
> > + if (p-buf+221 > alloced) {
> >   int len=p-buf;
> >   char *buf2;
> >   alloced += PAGE_SIZE;
> > @@ -2629,8 +2629,8 @@ static int print_sigpending_alloc(char **bufp, struct sigpending
*pending)
> >   p = buf+len;
> >   }
> >
> > - p += sprintf(p, "sig %d: user %d flags %d",
> > -   info->si_signo, (int)q->user->uid, q->flags);
> > + p += sprintf(p, "sig %d: uid %d pid %d flags %d",
> > +   info->si_signo, (int)q->user->uid, info->si_pid, q->flags);
> >   p += sprintf(p, " errno %d code %d\n",
> >   info->si_errno, info->si_code);
> >
> > @@ -2739,3 +2739,215 @@ int task_read_procsig(struct task_struct *p, char *name, char

```
**value)
> >
> >   return ret;
> >  }
> > +
> > +static int set_sigset(sigset_t *sig, char *value, int size)
> > +{
> > + int i;
> > +
> > + if (size < _NSIG)
> > +  return -EINVAL;
> > + sigemptyset(sig);
> > + for (i=0; i<_NSIG; i++)
> > +  if (value[i] == '1')
> > +   sigaddset(sig, i);
> > + return 0;
> > +}
> > +
> > +static char *next_eol(char *s, int maxlen)
> > +{
> > + int len=0;
> > + while (len<maxlen && *s!='\0' && *s !='\n')
> > +  s++, len++;
> > + if (len < maxlen)
> > +  return s;
> > + return NULL;
> > +}
> > +
> > +static int set_sigpending(struct task_struct *t, struct sigpending *pending,
> > +      char *value, int size)
> > +{
> > + struct sigqueue *q;
> > + int ret;
> > +#ifdef __ARCH_SI_TRAPNO
> > + int trapno;
> > +#endif
> > +
> > + if (set_sigset(&pending->signal, value, size))
> > +  return -EINVAL;
> > +
> > + size -= _NSIG+1;
> > + value += _NSIG+1;
> > + while (size) {
> > +  int signum, uid, pid, flags, errno, code, status, fd;
> > +  unsigned long utime, stime, addr, band;
> > +  char *start, *eol = next_eol(value, size);
> > +
> > +  if (!eol)
```

```
> > +   return 0;
> > +
> > +  size -= (eol - value) + 1;
> > +  start = eol+1;
> > +  ret = sscanf(value, "sig %d: uid %d pid %d flags %d errno %d code %d\n",
> > +   &signum, &uid, &pid, &flags, &errno, &code);
> > +  if (ret != 6)
> > +   goto out;
> > +  q = __sigqueue_alloc(t, GFP_KERNEL, 1);
> > +  if (!q)
> > +   goto out;
> > +  q->info.si_signo = signum;
> > +  q->info.si_errno = errno;
> > +  q->info.si_code = code;
> > +  q->info.si_uid = uid;
> > +  q->info.si_pid = pid;
> > +
> > +  switch(signum) {
> > +   /* XXX skipping posix1b timers and signals for now */
> > +   default: break;
> > +   case SIGKILL:
> > +    eol = next_eol(start, size);
> > +    if (!eol)
> > +     goto out;
> > +    size -= (eol - start) + 1;
> > +    ret = sscanf(start, " pid %d uid %d\n", &pid, &uid);
> > +    if (ret != 2)
> > +     goto out;
> > +    q->info._sifields._kill._pid = pid;
> > +    q->info._sifields._kill._uid = uid;
> > +    break;
> > +
> > +   case SIGCHLD:
> > +    eol = next_eol(start, size);
> > +    if (!eol)
> > +     goto out;
> > +    size -= (eol - start) + 1;
> > +    ret = sscanf(start, "pid %d uid %d status %d utime %lu stime %lu\n",
> > +     &pid, &uid, &status, &utime, &stime);
> > +    if (ret != 5)
> > +     goto out;
> > +    q->info._sifields._sigchld._pid = pid;
> > +    q->info._sifields._sigchld._uid = uid;
> > +    q->info._sifields._sigchld._status = status;
> > +    q->info._sifields._sigchld._utime = utime;
> > +    q->info._sifields._sigchld._stime = stime;
> > +    break;
> > +
```

```
> > +   case SIGILL:
> > +   case SIGFPE:
> > +   case SIGSEGV:
> > +   case SIGBUS:
> > +    eol = next_eol(start, size);
> > +    if (!eol)
> > +     goto out;
> > +    size -= (eol - start) + 1;
> > +#ifdef __ARCH_SI_TRAPNO
> > +    ret = sscanf(start, " addr %lu trapno %d\n", &addr, &trapno);
> > +    if (ret != 2)
> > +     goto out;
> > +    q->info._sifields._sigfault._trapno = trapno;
> > +#else
> > +    ret = sscanf(start, " addr %lu\n", &addr);
> > +    if (ret != 1)
> > +     goto out;
> > +#endif
> > +    q->info._sifields._sigfault._addr = (void __user *)addr;
> > +    break;
> > +
> > +   case SIGPOLL:
> > +    eol = next_eol(start, size);
> > +    if (!eol)
> > +     goto out;
> > +    size -= (eol - start) + 1;
> > +    ret = sscanf(start, " band %ld fd %d\n", &band, &fd);
> > +    if (ret != 2)
> > +     goto out;
> > +    q->info._sifields._sigpoll._band = band;
> > +    q->info._sifields._sigpoll._fd = fd;
> > +    break;
> > +  }
> > +  start = eol+1;
> > +  list_add_tail(&q->list, &pending->list);
> > + }
> > +
> > +out:
> > + return size;
> > +}
> > +
> > +static int set_sigaction_list(struct sighand_struct *sighand, char *value,
> > +    int size)
> > +{
> > + struct k_sigaction *action;
> > + char *eol;
> > + int num, ret;
> > + int origsize = size;
```

```
> > + //char sa_mask[_NSIG+1];
> > + char sa_mask[65];
> > + unsigned long sa_flags, sa_handler;
> > +
> > + while (size > 0 && (eol=next_eol(value, size))) {
> > +  sa_mask[64] = '\0';
> > +  ret = sscanf(value, "%d %lu %64s %lu\n",
> > +   &num, &sa_flags, sa_mask, &sa_handler);
> > +  if (ret != 4)
> > +   break;
> > +  if (num < 0 || num > _NSIG)
> > +   return -EINVAL;
> > +  action = &sighand->action[num];
> > +  action->sa.sa_flags = sa_flags;
> > +  set_sigset(&action->sa.sa_mask, sa_mask, _NSIG);
> > +  action->sa.sa_handler = (__sighandler_t) sa_handler;
> > +  size -= eol-value+1;
> > +  value = eol+1;
> > + }
> > +
> > + return origsize - size;
> > +}
> > +
> > +static int set_altstack(struct task_struct *p, char *value, int size)
> > +{
> > + unsigned long sas_ss_sp;
> > + size_t sas_ss_size;
> > + int ret;
> > +
> > + ret = sscanf(value, "%lu %zd\n", &sas_ss_sp, &sas_ss_size);
> > + if (ret != 2)
> > +  return -EINVAL;
> > + p->sas_ss_sp = sas_ss_sp;
> > + p->sas_ss_size = sas_ss_size;
> > +
> > + return size;
> > +}
> > +
> > +int task_write_procsig(struct task_struct *p, char *name, void *value,
> > +   size_t size)
> > +{
> > + int ret;
> > +
> > + if (current != p) {
> > +  ret = security_ptrace(current, p);
> > +  if (ret)
> > +   return ret;
> > + }
```

```
> > +
> > + ret = -EINVAL;
> > +
> > + if (strcmp(name, "pending") == 0) {
> > +  /* not masking out blocked signals  yet */
> > +  ret = set_sigpending(p, &p->pending, value, size);
> > + } else if (strcmp(name, "shared_pending") == 0) {
> > +  ret = set_sigpending(p, &p->signal->shared_pending, value, size);
> > + } else if (strcmp(name, "blocked") == 0) {
> > +  /* not masking out blocked signals  yet */
> > +  ret = set_sigset(&p->blocked, value, size);
> > + } else if (strcmp(name, "action") == 0) {
> > +  ret = set_sigaction_list(p->sighand, value, size);
> > + }  else if (strcmp(name, "altstack") == 0) {
> > +  ret = set_altstack(p, value, size);
> > +#if 0
> > + } else if (strcmp(name, "waiters") == 0) {
> > +  /* reset this at the waiter's restart using do_wait */
> > +  ret = set_wait_info(p->signal, value,  size);
> > +#endif
> > + }
> > +
> > + return ret;
> > +}
```

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by serue on Mon, 11 Jun 2007 17:05:37 GMT
View Forum Message <> Reply to Message

Quoting Cedric Le Goater (clg@fr.ibm.com):
> Serge E. Hallyn wrote:
> > As I mentioned earlier, I don't know what sort of approach we want
> > to take to guide checkpoint and restart. I.e. do we want it to be
> > a mostly userspace-orchestrated affair, or entirely done in the
> > kernel using the freezer or some other mechanism in response to a
> > single syscall or containerfs file write?
> >
> > If we wanted to do a lot of the work in userspace, here is a pair of
> > patches to read and restore signal information. It's entirely unsafe
> > wrt locking, bc i would assume that if we did in fact do c/r from
> > userspace, we would have some way of entirely pulling the task off
> > the runqueue while doing our thing...
> >

> > Anyway, this is purely to start discussion.
> >
> > thanks,
> > -serge
> >
> >>From 30bbe322942e5ed86bad63861dad80595cd04063 Mon Sep 17 00:00:00 2001
> > From: Serge E. Hallyn <serue@us.ibm.com>
> > Date: Mon, 30 Apr 2007 16:22:44 -0400
> > Subject: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
> >
> > Define /proc/<pid>/sig/ directory containing files to report
> > on a process' signal info.
> >
> > Files defined:
> > action: list signal action
> > altstack: print sigaltstack location and size
> > blocked: print blocked signal mask
> > pending: print pending signals and siginfo
> > shared_pending: print shared pending signals and siginfo
> > waiters: list tasks wait4()ing on task PID.
>
> some of these are already in /proc/<pid>/status and /proc/<pid>/stat

Good point. However they can't be set through that file. So is it
worth making the files write-only, to avoid duplication, or do we prefer
to not make the checkpoint and restart use different files?

> should we continue to use /proc ? or switch to some other mechanisms
> like getnetlink (taskstats) to map kernel structures.

We want to avoid 'map'ping kernel structures, though, right? We can
dump the data in a more generic fashion through netlink, dunno what we
prefer. But this is very definately process information :), so /proc
does seem appropriate.

> > +/*
> > + * print a sigset_t to a buffer. Return # characters printed,
> > + * not including the final ending '\0'.
> > + */
> > +static int print_sigset(char *buf, sigset_t *sig)
> > +{
> > + int i;
> > +
> > + for (i=0; i<_NSIG; i++) {
> > + if (sigismember(sig, i))
> > +   buf[i] = '1';
> > + else
> > +   buf[i] = '0';

> > + }
> > + buf[_NSIG] = '\0';
> > +
> > + return _NSIG;
> > +}
>
> you might want to use render_sigset_t() in fs/proc array.C

Thanks. (I will incorporate improvements like this if i end up sending another version)

> > +static int print_sigset_alloc(char **bufp, sigset_t *sig)
> > +{
> > + char *buf;
> > +
> > + *bufp = buf = kmalloc(_NSIG+1, GFP_KERNEL);
> > + if (!buf)
> > + return -ENOMEM;
> > +
> > + return print_sigset(buf, sig);
> > +}
> > +
> > +static int print_wait_info(char **bufp, struct signal_struct *signal)
> > +{
> > + char *buf;
> > + wait_queue_t *wait;
> > +
> > + *bufp = buf = kmalloc(PAGE_SIZE, GFP_KERNEL);
> > + if (!buf)
> > + return -ENOMEM;
> > +
> > + spin_lock(&signal->wait_chldexit.lock);
> > +
> > + list_for_each_entry(wait, &signal->wait_chldexit.task_list, task_list) {
> > + struct task_struct *tsk = wait->private;
> > +
> > + if (buf - *bufp +50 > PAGE_SIZE) {
> > + spin_unlock(&signal->wait_chldexit.lock);
> > + kfree(buf);
> > + *bufp = NULL;
> > + return -ENOMEM;
> > + }
> > + WARN_ON(wait->func != default_wake_function);
> > + buf += sprintf(buf, "%u %d\n", wait->flags, tsk->pid);
> > + }
> > +
> > + spin_unlock(&signal->wait_chldexit.lock);
> > +

```
> > + return buf-*bufp;
> > +}
> > +
> > +static int print_sigpending_alloc(char **bufp, struct sigpending *pending)
> > +{
> > + int alloced=0;
> > + char *buf, *p;
> > + struct sigqueue *q;
> > + struct siginfo *info;
> > +
> > + alloced = PAGE_SIZE;
> > + p = buf = kmalloc(alloced, GFP_KERNEL);
> > + if (!buf)
> > +  return -ENOMEM;
> > +
> > + p += print_sigset(buf, &pending->signal);
> > + p += sprintf(p, "\n");
> > +
> > + list_for_each_entry(q, &pending->list, list) {
> > +  info = &q->info;
> > +  if (p-buf+215 > alloced) {
> > +   int len=p-buf;
> > +   char *buf2;
> > +   alloced += PAGE_SIZE;
> > +   buf2 = kmalloc(alloced, GFP_KERNEL);
> > +   if (!buf2) {
> > +    kfree(buf);
> > +    return -ENOMEM;
> > +   }
> > +   memcpy(buf2, buf, alloced - PAGE_SIZE);
> > +   kfree(buf);
> > +   buf = buf2;
> > +   p = buf+len;
> > +  }
> > +
> > +  p += sprintf(p, "sig %d: user %d flags %d",
> > +   info->si_signo, (int)q->user->uid, q->flags);
> > +  p += sprintf(p, " errno %d code %d\n",
> > +   info->si_errno, info->si_code);
> > +
> > +  switch(info->si_signo) {
> > +  case SIGKILL:
> > +   p += sprintf(p, "  spid %d suid %d\n",
> > +    info->_sifields._kill._pid,
> > +    info->_sifields._kill._uid);
> > +   break;
> > +  /* XXX skipping posix1b timers and signals for now */
> > +  case SIGCHLD:
```

> > +   p += sprintf(p, " pid %d uid %d status %d utime %lu stime %lu\n",
> > +   info->_sifields._sigchld._pid,
> > +   info->_sifields._sigchld._uid,
> > +   info->_sifields._sigchld._status,
> > +   info->_sifields._sigchld._utime,
> > +   info->_sifields._sigchld._stime);
> > +   break;
> > + case SIGILL:
> > + case SIGFPE:
> > + case SIGSEGV:
> > + case SIGBUS:
> > +#ifdef __ARCH_SI_TRAPNO
> > +   p += sprintf(p, " addr %lu trapno %d\n",
> > +   (unsigned long)info->_sifields._sigfault._addr,
> > +   info->_sifields._sigfault._trapno);
> > +#else
> > +   p += sprintf(p, " addr %lu\n",
> > +   (unsigned long)info->_sifields._sigfault._addr);
> > +#endif
> > +   break;
> > + case SIGPOLL:
> > +   p += sprintf(p, " band %ld fd %d\n",
> > +   (long)info->_sifields._sigpoll._band,
> > +   info->_sifields._sigpoll._fd);
> > +   break;
> > + default:
> > +   p += sprintf(p, " Unsupported siginfo for signal %d\n",
> > +   info->si_signo);
> > +   break;
> > + }
> > + }
> > + *bufp = buf;
> > + return p-buf;
> > +}
>
> I think we are reaching the limit of /proc when we expose the pending siginfos.

Why?

-serge

> > +static int print_sigaction_list(char **bufp, struct sighand_struct *sighand)
> > +{
> > + struct k_sigaction *action;
> > + int maxlen;
> > + int i;
> > + char *buf;
> > +

```
> > + /* two unsigned longs (20 chars), one int (10 chars), a sigset_t, 3 spaces, plus a newline */
> > + maxlen = 10 + 20*2 + _NSIG + 4;
> > + /* and we have _NSIG of those entries, plus an ending \0 */
> > + maxlen *= _NSIG+1;
> > +
> > + *bufp = buf = kmalloc(maxlen, GFP_KERNEL);
> > + if (!buf)
> > +  return -ENOMEM;
> > +
> > + spin_lock(&sighand->siglock);
> > + for (i=0; i<_NSIG; i++) {
> > +  action = &sighand->action[i];
> > +  buf += sprintf(buf, "%10d %20lu ", i, action->sa.sa_flags);
> > +  buf += print_sigset(buf, &action->sa.sa_mask);
> > +  buf += sprintf(buf, " %20lu\n", (unsigned long)action->sa.sa_handler);
> > +  BUG_ON(buf-*bufp > maxlen);
> > + }
> > + spin_unlock(&sighand->siglock);
> > + return buf-*bufp;
> > +}
> > +
> > +int task_read_procsig(struct task_struct *p, char *name, char **value)
> > +{
> > + int ret;
> > +
> > + if (current != p) {
> > +  ret = security_ptrace(current, p);
> > +  if (ret)
> > +   return ret;
> > + }
> > +
> > + ret = -EINVAL;
> > +
> > + if (strcmp(name, "pending") == 0) {
> > +  /* not masking out blocked signals  yet */
> > +  ret = print_sigpending_alloc(value, &p->pending);
> > + } else if (strcmp(name, "blocked") == 0) {
> > +  /* not masking out blocked signals  yet */
> > +  ret = print_sigset_alloc(value, &p->blocked);
> > + } else if (strcmp(name, "shared_pending") == 0) {
> > +  ret = print_sigpending_alloc(value, &p->signal->shared_pending);
> > + } else if (strcmp(name, "waiters") == 0) {
> > +  ret = print_wait_info(value, p->signal);
> > + } else if (strcmp(name, "action") == 0) {
> > +  ret = print_sigaction_list(value, p->sighand);
> > + }  else if (strcmp(name, "altstack") == 0) {
> > +  *value = kmalloc(40, GFP_KERNEL);
> > +  ret = -ENOMEM;
```

> > + if (*value) {
> > +   ret = sprintf(*value, "%lu %zd", p->sas_ss_sp, p->sas_ss_size);
> > + }
> > + }
> > +
> > + return ret;
> > +}

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by Carl-Daniel Hailfinge on Tue, 12 Jun 2007 15:29:38 GMT
View Forum Message <> Reply to Message

On 11.06.2007 19:05, Serge E. Hallyn wrote:
> Quoting Cedric Le Goater (clg@fr.ibm.com):
>
>> should we continue to use /proc ? or switch to some other mechanisms
>> like getnetlink (taskstats) to map kernel structures.
>
> We want to avoid 'map'ping kernel structures, though, right? We can
> dump the data in a more generic fashion through netlink, dunno what we
> prefer. But this is very definately process information :), so /proc
> does seem appropriate.

While I agree that /proc seems appropriate, I see a few benefits of
dumping the data through netlink:
* Speed. IIRC there were benchmarks showing an advantage of netlink
  over /proc when communicating with userspace. Sorry, no idea where
  I read that.
* Versioning. While we strive to have the perfect interface on the
  first try, changes might be necessary. I see no way to handle
  multiple versions of an interface in /proc without big headaches.
* Conformity. With /proc, people often see a file, take a look at
  it and try to infer the structure of the file from what they see.
  This has led to multiple problems in the past when the content of
  some files in /proc changed slightly and tools broke. With
  netlink, implementers have to look at the spec to achieve anything
  useful.

Regards,
Carl-Daniel

_____
Containers mailing list
Containers@lists.linux-foundation.org

Subject: Re: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by Daniel Lezcano on Tue, 12 Jun 2007 16:02:40 GMT
View Forum Message <> Reply to Message

Carl-Daniel Hailfinger wrote:
> On 11.06.2007 19:05, Serge E. Hallyn wrote:
>> Quoting Cedric Le Goater (clg@fr.ibm.com):
>>
>>> should we continue to use /proc ? or switch to some other mechanisms
>>> like getnetlink (taskstats) to map kernel structures.
>> We want to avoid 'map'ping kernel structures, though, right? We can
>> dump the data in a more generic fashion through netlink, dunno what we
>> prefer. But this is very definately process information :), so /proc
>> does seem appropriate.
>
> While I agree that /proc seems appropriate, I see a few benefits of
> dumping the data through netlink:
> * Speed. IIRC there were benchmarks showing an advantage of netlink
>   over /proc when communicating with userspace. Sorry, no idea where
>   I read that.
> * Versioning. While we strive to have the perfect interface on the
>   first try, changes might be necessary. I see no way to handle
>   multiple versions of an interface in /proc without big headaches.
> * Conformity. With /proc, people often see a file, take a look at
>   it and try to infer the structure of the file from what they see.
>   This has led to multiple problems in the past when the content of
>   some files in /proc changed slightly and tools broke. With
>   netlink, implementers have to look at the spec to achieve anything
>   useful.

Right. And community seems to encourage to use the netlink and to stop
implementing new entry in /proc.

http://kerneltrap.org/node/6637

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by serue on Tue, 12 Jun 2007 16:12:56 GMT
View Forum Message <> Reply to Message

Quoting Carl-Daniel Hailfinger (c-d.hailfinger.devel.2006@gmx.net):
> On 11.06.2007 19:05, Serge E. Hallyn wrote:
> > Quoting Cedric Le Goater (clg@fr.ibm.com):
> >
> >> should we continue to use /proc ? or switch to some other mechanisms
> >> like getnetlink (taskstats) to map kernel structures.
> >
> > We want to avoid 'map'ping kernel structures, though, right? We can
> > dump the data in a more generic fashion through netlink, dunno what we
> > prefer. But this is very definately process information :), so /proc
> > does seem appropriate.
>
> While I agree that /proc seems appropriate, I see a few benefits of
> dumping the data through netlink:

Good points, thanks.

> * Speed. IIRC there were benchmarks showing an advantage of netlink
>   over /proc when communicating with userspace. Sorry, no idea where
>   I read that.

I don't think we're dumping large amounts of data (the largest amounts,
process memory, we're looking at doing just by forcing dump to swap), so
I'm not sure how much it matters.

Still,

> * Versioning. While we strive to have the perfect interface on the
>   first try, changes might be necessary. I see no way to handle
>   multiple versions of an interface in /proc without big headaches.

Good point, this kind of offsets my major point against netlink, that
we'd likely inherently end up versioning the interface by being tempted
to dump kernel structures verbatim. I doubt anyone would claim that
we'll never need to update the /proc interface, so that may make using
/proc a nonstarter.

> * Conformity. With /proc, people often see a file, take a look at
>   it and try to infer the structure of the file from what they see.
>   This has led to multiple problems in the past when the content of
>   some files in /proc changed slightly and tools broke. With
>   netlink, implementers have to look at the spec to achieve anything
>   useful.

Ok, so presumably we'd want some 'start a checkpoint' or 'start a
restore' command (through syscall or whatever) to create the netlink
socket and pass that to the various kernel dump/restore pieces?

Is there some better alternative people prefer to a syscall? If not,
Daniel, would you mind adding that to the front of your patchset, and
having your udp socket checkpoint/restore use that socket?

thanks,
-serge

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by Cedric Le Goater on Tue, 12 Jun 2007 16:30:46 GMT
View Forum Message <> Reply to Message

[...]

```
>>> +static int print_sigpending_alloc(char **bufp, struct sigpending *pending)
>>> +{
>>> + int alloced=0;
>>> + char *buf, *p;
>>> + struct sigqueue *q;
>>> + struct siginfo *info;
>>> +
>>> + alloced = PAGE_SIZE;
>>> + p = buf = kmalloc(alloced, GFP_KERNEL);
>>> + if (!buf)
>>> +  return -ENOMEM;
>>> +
>>> + p += print_sigset(buf, &pending->signal);
>>> + p += sprintf(p, "\n");
>>> +
>>> + list_for_each_entry(q, &pending->list, list) {
>>> +  info = &q->info;
>>> +  if (p-buf+215 > alloced) {
>>> +   int len=p-buf;
>>> +   char *buf2;
>>> +   alloced += PAGE_SIZE;
>>> +   buf2 = kmalloc(alloced, GFP_KERNEL);
>>> +   if (!buf2) {
>>> +    kfree(buf);
>>> +    return -ENOMEM;
>>> +   }
>>> +   memcpy(buf2, buf, alloced - PAGE_SIZE);
>>> +   kfree(buf);
>>> +   buf = buf2;
>>> +   p = buf+len;
```

```
>>> +  }
>>> +
>>> +  p += sprintf(p, "sig %d: user %d flags %d",
>>> +   info->si_signo, (int)q->user->uid, q->flags);
>>> +  p += sprintf(p, " errno %d code %d\n",
>>> +   info->si_errno, info->si_code);
>>> +
>>> +  switch(info->si_signo) {
>>> +  case SIGKILL:
>>> +   p += sprintf(p, "  spid %d suid %d\n",
>>> +    info->_sifields._kill._pid,
>>> +    info->_sifields._kill._uid);
>>> +   break;
>>> +  /* XXX skipping posix1b timers and signals for now */
>>> +  case SIGCHLD:
>>> +   p += sprintf(p, "  pid %d uid %d status %d utime %lu stime %lu\n",
>>> +    info->_sifields._sigchld._pid,
>>> +    info->_sifields._sigchld._uid,
>>> +    info->_sifields._sigchld._status,
>>> +    info->_sifields._sigchld._utime,
>>> +    info->_sifields._sigchld._stime);
>>> +   break;
>>> +  case SIGILL:
>>> +  case SIGFPE:
>>> +  case SIGSEGV:
>>> +  case SIGBUS:
>>> +#ifdef __ARCH_SI_TRAPNO
>>> +   p += sprintf(p, "  addr %lu trapno %d\n",
>>> +    (unsigned long)info->_sifields._sigfault._addr,
>>> +    info->_sifields._sigfault._trapno);
>>> +#else
>>> +   p += sprintf(p, "  addr %lu\n",
>>> +    (unsigned long)info->_sifields._sigfault._addr);
>>> +#endif
>>> +   break;
>>> +  case SIGPOLL:
>>> +   p += sprintf(p, "  band %ld fd %d\n",
>>> +    (long)info->_sifields._sigpoll._band,
>>> +    info->_sifields._sigpoll._fd);
>>> +   break;
>>> +  default:
>>> +   p += sprintf(p, "  Unsupported siginfo for signal %d\n",
>>> +    info->si_signo);
>>> +   break;
>>> +  }
>>> + }
>>> + *bufp = buf;
>>> + return p-buf;
```

>>> +}
>> I think we are reaching the limit of /proc when we expose the pending siginfos.
>
> Why?

well, we are really exposing the internals of signal delivery, which are
not that useful for /proc. IMO, it would be better to use an "opaque" to
get/set the data we need.

C.

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 2/2] signal c/r: implement /proc/pid/sig writing
Posted by Cedric Le Goater on Tue, 12 Jun 2007 16:44:44 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Quoting Cedric Le Goater (clg@fr.ibm.com):
>> with all the infos you've gathered in /proc, why don't you just kill the
>> process ?
>>
>> The patch we have to restore pending signals in 2.6.21-mm2-lxc3 does :
>>
>> +static int pid_set_siginfo(mcrk_session_t * s, void *ptarg)
>> +{
>> + mcrk_pid_setsignal_t arg;
>> + siginfo_t si;
>> + int ret;
>> +
>> + if (!ptarg) {
>> + return -EINVAL;
>> + }
>> +
>> + if (copy_from_user(&arg, ptarg, sizeof(arg)))
>> + return -EFAULT;
>> + if (copy_from_user(&si, U64_2_PTR(arg.siginfo), sizeof(si)))
>> + return -EFAULT;
>
> Hmm, one problem with especially this second copy_from_user() is that
> you are making the checkpoint image more kernel dependant.

right. we need an opaque structure to hold the siginfo data.

> Whatever approach we take both high-level and low-level, we do want to
> avoid having checkpoint images directly reflect in-kernel structures,
> right?

yes.

> That's one area where the /proc approach has an inherent advantage over
> using netlink to dump information, it avoids the temptation to just dump
> and restore straight from the kernel pointer, which would threaten to
> make restoring a checkpoint from another kernel much more dangerous.

I agree. You have to be self disciplined and define nice structures for
all the data you want to exchange between kernel and user.

>> + if (arg.shared) {
>> + ret = kill_proc_info(si.si_signo, &si, current->pid);
>> + } else {
>> + ret = send_sig_info(si.si_signo, &si, current);
>> + }
>> + return ret;
>> +}
>
> This part is fine with me, but assumes we take the more kernel-guided
> approach, right.
>
> And that's what I'm trying to get people to discuss :) Do we want a
> more kernel-guided approach, or do we want to provide pieces of
> functionality that userspace exploits?
>
> Oh, or are you saying this would just replace the biggest chunk of my
> set_sigpending() function below?

I think so :)

cheers,

C.

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by serue on Tue, 12 Jun 2007 16:55:55 GMT
View Forum Message <> Reply to Message

Quoting Daniel Lezcano (dlezcano@fr.ibm.com):
> Carl-Daniel Hailfinger wrote:
> >On 11.06.2007 19:05, Serge E. Hallyn wrote:
> >>Quoting Cedric Le Goater (clg@fr.ibm.com):
> >>
> >>>should we continue to use /proc ? or switch to some other mechanisms
> >>>like getnetlink (taskstats) to map kernel structures.
> >>We want to avoid 'map'ping kernel structures, though, right? We can
> >>dump the data in a more generic fashion through netlink, dunno what we
> >>prefer. But this is very definately process information :), so /proc
> >>does seem appropriate.
> >
> >While I agree that /proc seems appropriate, I see a few benefits of
> >dumping the data through netlink:
> >* Speed. IIRC there were benchmarks showing an advantage of netlink
> > over /proc when communicating with userspace. Sorry, no idea where
> > I read that.
> >* Versioning. While we strive to have the perfect interface on the
> > first try, changes might be necessary. I see no way to handle
> > multiple versions of an interface in /proc without big headaches.
> >* Conformity. With /proc, people often see a file, take a look at
> > it and try to infer the structure of the file from what they see.
> > This has led to multiple problems in the past when the content of
> > some files in /proc changed slightly and tools broke. With
> > netlink, implementers have to look at the spec to achieve anything
> > useful.
>
> Right. And community seems to encourage to use the netlink and to stop
> implementing new entry in /proc.
>
> http://kerneltrap.org/node/6637

That's not quite what that thread is saying :)

For just this information, I would prefer /proc over netlink. But since
we'll be dumping a whole bunch more data, I agree netlink may be the way
to go.

thanks,
-serge

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/

Serge E. Hallyn wrote:
> Quoting Carl-Daniel Hailfinger (c-d.hailfinger.devel.2006@gmx.net):
>> On 11.06.2007 19:05, Serge E. Hallyn wrote:
>>> Quoting Cedric Le Goater (clg@fr.ibm.com):
>>>
>>>> should we continue to use /proc ? or switch to some other mechanisms
>>>> like getnetlink (taskstats) to map kernel structures.
>>> We want to avoid 'map'ping kernel structures, though, right? We can
>>> dump the data in a more generic fashion through netlink, dunno what we
>>> prefer. But this is very definately process information :), so /proc
>>> does seem appropriate.
>> While I agree that /proc seems appropriate, I see a few benefits of
>> dumping the data through netlink:
>
> Good points, thanks.
>
>> * Speed. IIRC there were benchmarks showing an advantage of netlink
>> over /proc when communicating with userspace. Sorry, no idea where
>> I read that.
>
> I don't think we're dumping large amounts of data (the largest amounts,
> process memory, we're looking at doing just by forcing dump to swap), so
> I'm not sure how much it matters.
>
> Still,
>
>> * Versioning. While we strive to have the perfect interface on the
>> first try, changes might be necessary. I see no way to handle
>> multiple versions of an interface in /proc without big headaches.
>
> Good point, this kind of offsets my major point against netlink, that
> we'd likely inherently end up versioning the interface by being tempted
> to dump kernel structures verbatim. I doubt anyone would claim that
> we'll never need to update the /proc interface, so that may make using
> /proc a nonstarter.
>
>> * Conformity. With /proc, people often see a file, take a look at
>> it and try to infer the structure of the file from what they see.
>> This has led to multiple problems in the past when the content of
>> some files in /proc changed slightly and tools broke. With
>> netlink, implementers have to look at the spec to achieve anything
>> useful.
>
> Ok, so presumably we'd want some 'start a checkpoint' or 'start a
> restore' command (through syscall or whatever) to create the netlink
> socket and pass that to the various kernel dump/restore pieces?

>
> Is there some better alternative people prefer to a syscall? If not,
> Daniel, would you mind adding that to the front of your patchset, and
> having your udp socket checkpoint/restore use that socket?

I am not sure I understand what you want. Knowing I talk english like a
french cow, perhaps I missed something. Just let me know ...

The udp socket c/r is:
  - you provide a fd, you get a raw data (for statefile).
  - you provide a raw data, you get a fd.

The generic netlink are on top of the netlinks. You subscribe to a
family (in this case AF_INET_CR), you send a message with the DUMP
command and the fd parameter of the socket you want to checkpoint and
you read the dump data. If you want to restore it, you send the message
with the RESTORE command and the attributes you received from the
previous dump message and you read the answer which contains the fd of
the newly created socket.

What you are proposing is to create a syscall for restore and
checkpoint. If the generic netlink is used, this is useless, because you
can just create a CHECKPOINT/RESTORE command message and use
socket/write/read/close to get all the statefile.

But if this approach is choosen, then that means *all* kernel resources
should be passed to the netlink message. The netlink message attributes
will need to be defined for all the differents internal kernel data.
This means thousands of attributes, so impossible to do. The turn-around
in this case is to pass a raw binary attribute but we lose the netlink
advantages and we fall into a "/proc" approach.

IMHO, one approach could be:
  - classify the resources to be checkpointed
  - define a family for each resources
  - define the message attributes for each resources
  - find in which order to restore resources (eg. shared memory should
be restored first and after sem undos can be restored in a process context).

If we are able to have a small application checkpointing itself, using
the netlink mechanism. That can be a very first to step before choosing
to checkpoint/restart from userspace or kernelspace or both.

   -- Daniel

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by Daniel Lezcano on Thu, 14 Jun 2007 14:38:07 GMT
View Forum Message <> Reply to Message

Serge E. Hallyn wrote:
> Quoting Carl-Daniel Hailfinger (c-d.hailfinger.devel.2006@gmx.net):
>> On 11.06.2007 19:05, Serge E. Hallyn wrote:
>>> Quoting Cedric Le Goater (clg@fr.ibm.com):
>>>
>>>> should we continue to use /proc ? or switch to some other mechanisms
>>>> like getnetlink (taskstats) to map kernel structures.
>>> We want to avoid 'map'ping kernel structures, though, right? We can
>>> dump the data in a more generic fashion through netlink, dunno what we
>>> prefer. But this is very definately process information :), so /proc
>>> does seem appropriate.
>> While I agree that /proc seems appropriate, I see a few benefits of
>> dumping the data through netlink:
>
> Good points, thanks.
>
>> * Speed. IIRC there were benchmarks showing an advantage of netlink
>> over /proc when communicating with userspace. Sorry, no idea where
>> I read that.
>
> I don't think we're dumping large amounts of data (the largest amounts,
> process memory, we're looking at doing just by forcing dump to swap), so
> I'm not sure how much it matters.
>
> Still,
>
>> * Versioning. While we strive to have the perfect interface on the
>> first try, changes might be necessary. I see no way to handle
>> multiple versions of an interface in /proc without big headaches.
>
> Good point, this kind of offsets my major point against netlink, that
> we'd likely inherently end up versioning the interface by being tempted
> to dump kernel structures verbatim. I doubt anyone would claim that
> we'll never need to update the /proc interface, so that may make using
> /proc a nonstarter.
>
>> * Conformity. With /proc, people often see a file, take a look at
>> it and try to infer the structure of the file from what they see.
>> This has led to multiple problems in the past when the content of
>> some files in /proc changed slightly and tools broke. With
>> netlink, implementers have to look at the spec to achieve anything
>> useful.
>
> Ok, so presumably we'd want some 'start a checkpoint' or 'start a
> restore' command (through syscall or whatever) to create the netlink

> socket and pass that to the various kernel dump/restore pieces?
>
> Is there some better alternative people prefer to a syscall? If not,
> Daniel, would you mind adding that to the front of your patchset, and
> having your udp socket checkpoint/restore use that socket?

Sorry, resend with the right <from> email.

I am not sure I understand what you want. Knowing I talk english like a
french cow, perhaps I missed something. Just let me know ...

The udp socket c/r is:
  - you provide a fd, you get a raw data (for statefile).
  - you provide a raw data, you get a fd.

The generic netlink are on top of the netlinks. You subscribe to a
family (in this case AF_INET_CR), you send a message with the DUMP
command and the fd parameter of the socket you want to checkpoint and
you read the dump data. If you want to restore it, you send the message
with the RESTORE command and the attributes you received from the
previous dump message and you read the answer which contains the fd of
the newly created socket.

What you are proposing is to create a syscall for restore and
checkpoint. If the generic netlink is used, this is useless, because you
can just create a CHECKPOINT/RESTORE command message and use
socket/write/read/close to get all the statefile.

But if this approach is choosen, then that means *all* kernel resources
should be passed to the netlink message. The netlink message attributes
will need to be defined for all the differents internal kernel data.
This means thousands of attributes, so impossible to do. The turn-around
in this case is to pass a raw binary attribute but we lose the netlink
advantages and we fall into a "/proc" approach.

IMHO, one approach could be:
  - classify the resources to be checkpointed
  - define a family for each resources
  - define the message attributes for each resources
  - find in which order to restore resources (eg. shared memory should
be restored first and after sem undos can be restored in a process context).

If we are able to have a small application checkpointing itself, using
the netlink mechanism. That can be a very first to step before choosing
to checkpoint/restart from userspace or kernelspace or both.

   -- Daniel

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 1/2] signal checkpoint: define /proc/pid/sig/
Posted by serue on Mon, 02 Jul 2007 21:40:40 GMT
View Forum Message <> Reply to Message

Quoting Daniel Lezcano (dlezcano@fr.ibm.com):
> Serge E. Hallyn wrote:
> >Quoting Carl-Daniel Hailfinger (c-d.hailfinger.devel.2006@gmx.net):
> >>On 11.06.2007 19:05, Serge E. Hallyn wrote:
> >>>Quoting Cedric Le Goater (clg@fr.ibm.com):
> >>>
> >>>>should we continue to use /proc ? or switch to some other mechanisms
> >>>>like getnetlink (taskstats) to map kernel structures.
> >>>We want to avoid 'map'ping kernel structures, though, right? We can
> >>>dump the data in a more generic fashion through netlink, dunno what we
> >>>prefer. But this is very definately process information :), so /proc
> >>>does seem appropriate.
> >>While I agree that /proc seems appropriate, I see a few benefits of
> >>dumping the data through netlink:
> >
> >Good points, thanks.
> >
> >>* Speed. IIRC there were benchmarks showing an advantage of netlink
> >> over /proc when communicating with userspace. Sorry, no idea where
> >> I read that.
> >
> >I don't think we're dumping large amounts of data (the largest amounts,
> >process memory, we're looking at doing just by forcing dump to swap), so
> >I'm not sure how much it matters.
> >
> >Still,
> >
> >>* Versioning. While we strive to have the perfect interface on the
> >> first try, changes might be necessary. I see no way to handle
> >> multiple versions of an interface in /proc without big headaches.
> >
> >Good point, this kind of offsets my major point against netlink, that
> >we'd likely inherently end up versioning the interface by being tempted
> >to dump kernel structures verbatim. I doubt anyone would claim that
> >we'll never need to update the /proc interface, so that may make using
> >/proc a nonstarter.
> >
> >>* Conformity. With /proc, people often see a file, take a look at

> >> it and try to infer the structure of the file from what they see.
> >> This has led to multiple problems in the past when the content of
> >> some files in /proc changed slightly and tools broke. With
> >> netlink, implementers have to look at the spec to achieve anything
> >> useful.
> >
> >Ok, so presumably we'd want some 'start a checkpoint' or 'start a
> >restore' command (through syscall or whatever) to create the netlink
> >socket and pass that to the various kernel dump/restore pieces?
> >
> >Is there some better alternative people prefer to a syscall? If not,
> >Daniel, would you mind adding that to the front of your patchset, and
> >having your udp socket checkpoint/restore use that socket?
>
> Sorry, resend with the right <from> email.
>
> I am not sure I understand what you want. Knowing I talk english like a
> french cow, perhaps I missed something. Just let me know ...

Mooooo.

> The udp socket c/r is:
> - you provide a fd, you get a raw data (for statefile).
> - you provide a raw data, you get a fd.
>
> The generic netlink are on top of the netlinks. You subscribe to a
> family (in this case AF_INET_CR), you send a message with the DUMP
> command and the fd parameter of the socket you want to checkpoint and
> you read the dump data. If you want to restore it, you send the message
> with the RESTORE command and the attributes you received from the
> previous dump message and you read the answer which contains the fd of
> the newly created socket.
>
> What you are proposing is to create a syscall for restore and
> checkpoint. If the generic netlink is used, this is useless, because you
> can just create a CHECKPOINT/RESTORE command message and use
> socket/write/read/close to get all the statefile.

I was just thinking of consolidating all the genetlink stuff so we don't
have N genetlink families, one for each checkpointable subsystem.

But it doesn't matter. After I finally started trying to implement the
signal c/r on top of netlink instead of /proc, it occurred to me there
is the same problem as we had with audit on netlink, except without
as simple a possibility of resolution (afaics) bc of the lack of pidns
ids.

The problem is that a netlink send and it's handler are completely

asynchronous to each other. So let's say the user sends in the pid of the process whose signal state should be checkpointed. It presumably does an rtnetlink_send to send the pid, but if you trace what happens to that data, it gets placed on the skb queue and then pulled off by a handler, which could be in a bogus pid namespace. Since we have no way of specifying pid namespaces, this means we can't actually disambiguate tasks with the same pid number.

One workaround would be to use the ids which are introduced by Cedric's bind_ns patchset.

Or, we can stick with /proc interface for signal checkpoint and restart.

Or, use a container subsystem to walk the list of tasks in the container and checkpoint signal info for all the tasks. Could be combined into the freezer subsystem, into a separate checkpoint_restart subsystem combined with all other things to be checkpointed, or just made into it's own separate subsystem just for signal info.

Any other ideas?

thanks,
-serge

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers